**重庆邮电大学移通学院**

**ACM模板总汇**

**版本：0.9\_Alpha**

**修改时间：2016-11-11 11:35:17**

**By: xbh, lyr**

目录

[头文件 6](#_Toc466627649)

[快速读入 7](#_Toc466627650)

[快速输出 8](#_Toc466627651)

[---------图论： 9](#_Toc466627652)

[弦图判断 9](#_Toc466627653)

[快速最大团 10](#_Toc466627654)

[求最大团的点数 12](#_Toc466627655)

[二分图匹配的匈牙利算法 13](#_Toc466627656)

[基于spfa的最小费用流 14](#_Toc466627657)

[ISAP 求最大流 最小割 15](#_Toc466627658)

[Dinic 求最大流 最小割 18](#_Toc466627659)

[求最大匹配、最小覆盖 20](#_Toc466627660)

[稳定婚姻问题 22](#_Toc466627661)

[求二分图最佳完美匹配的KM算法 24](#_Toc466627662)

[最小树形图 25](#_Toc466627663)

[最近公共祖先 27](#_Toc466627664)

[2-SAT 29](#_Toc466627665)

[Tarjan的求点-双连通分量算法 30](#_Toc466627666)

[Tarjan求强连通分量 32](#_Toc466627667)

[无向图的割顶和桥判断 32](#_Toc466627668)

[二分图黑白着色 33](#_Toc466627669)

[拓扑排序 34](#_Toc466627670)

[Kruskal求最小生成树 35](#_Toc466627671)

[Dijkstra(pair\_heap\_tag优化) 36](#_Toc466627672)

[普通版本的Dijkstra 37](#_Toc466627673)

[Bellman Ford 39](#_Toc466627674)

[Minimal Steiner Tree 41](#_Toc466627675)

[Floyd Warshall(求每两点之间的最短路) 43](#_Toc466627676)

[Floyd 求最小环 43](#_Toc466627677)

[------------dp相关： 45](#_Toc466627678)

[最大子串匹配 45](#_Toc466627679)

[最大子段和 46](#_Toc466627680)

[最大子阵和 46](#_Toc466627681)

[最长单调子序列 47](#_Toc466627682)

[序列逆序对数 48](#_Toc466627683)

[最长公共递增子序列 49](#_Toc466627684)

[------数学： 50](#_Toc466627685)

[清华定积分 50](#_Toc466627686)

[上交定积分 51](#_Toc466627687)

[牛顿法解多项式的根 51](#_Toc466627688)

[矩阵乘法 52](#_Toc466627689)

[求n以内的所有素数 54](#_Toc466627690)

[唯一分解(e[i]为p[i]的个数) 54](#_Toc466627691)

[欧几里德 54](#_Toc466627692)

[扩展欧几里德 ax+by=d, d = gcd(a, b) 55](#_Toc466627693)

[计算模n下a的逆，不存在返回1 55](#_Toc466627694)

[大整数取模 55](#_Toc466627695)

[幂取模，返回a^p mod n(0 <= a < n) 55](#_Toc466627696)

[中国剩余定理 56](#_Toc466627697)

[求解模方程 a^x = b(mod n)。n为素数，无解返回-1 56](#_Toc466627698)

[求二项式定理中各项的系数 56](#_Toc466627699)

[求c(i, j) 57](#_Toc466627700)

[n的欧拉函数值(不超过n且与n互素的正整数个数) 57](#_Toc466627701)

[1~n中所有数的欧拉函数值 57](#_Toc466627702)

[Pell方程 57](#_Toc466627703)

[求x^2 mod p = a 的解 59](#_Toc466627704)

[线性规划（单纯形法） 59](#_Toc466627705)

[---------数据结构： 61](#_Toc466627706)

[带花树 61](#_Toc466627707)

[SBT 62](#_Toc466627708)

[Treap 65](#_Toc466627709)

[Trie字典树 67](#_Toc466627710)

[Splay 69](#_Toc466627711)

[并查集 72](#_Toc466627712)

[RMQ问题（Tarjan的Sparse-Table算法） 72](#_Toc466627713)

[二叉索引树（树状数组）二维 73](#_Toc466627714)

[基于二分的离散化 73](#_Toc466627715)

[--------其他： 74](#_Toc466627716)

[最少找硬币 74](#_Toc466627717)

[棋盘分割 75](#_Toc466627718)

[汉罗塔 76](#_Toc466627719)

[求阶乘最后非零位是几 78](#_Toc466627720)

[约瑟夫环（n个人报数，报道m的出局，求最后一个人） 78](#_Toc466627721)

[N皇后构造解 79](#_Toc466627722)

[布尔母函数 79](#_Toc466627723)

[取第k小的元素 80](#_Toc466627724)

[幻方构造 80](#_Toc466627725)

[蔡勒（Zeller）公式 81](#_Toc466627726)

[直线下格点统计 81](#_Toc466627727)

[字符串的最小表示 82](#_Toc466627728)

[-----------大数与分数： 82](#_Toc466627729)

[分数的相关计算 82](#_Toc466627730)

[大整数（一） 83](#_Toc466627731)

[大整数（二） 86](#_Toc466627732)

[----------一些常见套路： 91](#_Toc466627733)

[2 台机器工作调度 91](#_Toc466627734)

[其他公式定理： 91](#_Toc466627735)

[部分知识点详解 93](#_Toc466627736)

[暴力的技巧 93](#_Toc466627737)

[一、01背包 93](#_Toc466627738)

[二、DLX详见DLX相关文件 93](#_Toc466627739)

[AC自动机 93](#_Toc466627740)

[字母树。 93](#_Toc466627741)

[AC自动机 95](#_Toc466627742)

[AC自动机常见套路 97](#_Toc466627743)

[错误注意事项 98](#_Toc466627744)

[附录完整模板 98](#_Toc466627745)

[快速傅里叶变换 102](#_Toc466627746)

[一、FFT的功能 102](#_Toc466627747)

[二、二维 FFT与二维卷积 103](#_Toc466627748)

[三、一维FFT与一维滤波器 103](#_Toc466627749)

[四、二维滤波器 104](#_Toc466627750)

[附录 105](#_Toc466627751)

[JAVA 110](#_Toc466627752)

[JAVA程序大体框架 110](#_Toc466627753)

[简单的大数例子 111](#_Toc466627754)

[二、BigDecimal大小数 112](#_Toc466627755)

[三、JAVA的一些STL小技巧 112](#_Toc466627756)

[KDTree 123](#_Toc466627757)

[简介 123](#_Toc466627758)

[KDTree的构建 123](#_Toc466627759)

[KDTree更新信息。 123](#_Toc466627760)

[询问操作 124](#_Toc466627761)

[插入与删除 124](#_Toc466627762)

[注意事项 125](#_Toc466627763)

[例题介绍 125](#_Toc466627764)

[KMP与扩展KMP与马拉车算法 129](#_Toc466627765)

[KMP：普通KMP 129](#_Toc466627766)

[EXKMP：扩展KMP 132](#_Toc466627767)

[Manacher算法介绍 133](#_Toc466627768)

[组合游戏与SG函数 134](#_Toc466627769)

[SG函数 134](#_Toc466627770)

[问题组合 134](#_Toc466627771)

[后缀数组 135](#_Toc466627772)

[板子 135](#_Toc466627773)

[实际效率： 139](#_Toc466627774)

[后缀数组的应用 139](#_Toc466627775)

[哈弗曼树 143](#_Toc466627776)

[一、哈弗曼树介绍 143](#_Toc466627777)

[二、多叉哈夫曼树 144](#_Toc466627778)

[三、多叉哈弗曼树公式 145](#_Toc466627779)

[四、多叉哈弗曼树构造法 145](#_Toc466627780)

[五、实现算法 145](#_Toc466627781)

[行列式与生成树计数 145](#_Toc466627782)

[行列式 145](#_Toc466627783)

[生成树计数 147](#_Toc466627784)

[树 152](#_Toc466627785)

[树的直径，树的最长链 152](#_Toc466627786)

[树链剖分 153](#_Toc466627787)

[简介 153](#_Toc466627788)

[部分代码细节讲解 154](#_Toc466627789)

[SPOJ的树剖代码 157](#_Toc466627790)

[树状数组： 161](#_Toc466627791)

[先上板子： 161](#_Toc466627792)

[一维： 163](#_Toc466627793)

[二维： 165](#_Toc466627794)

[网络流模型的一些建图 167](#_Toc466627795)

[分数规划 168](#_Toc466627796)

[最大权闭合图 169](#_Toc466627797)

[求解Maximize|Ei|-|Vi| 170](#_Toc466627798)

[最大密度子图 171](#_Toc466627799)

[有源汇，有下界的可行流 172](#_Toc466627800)

[有源汇，有下界最大流 172](#_Toc466627801)

[有源汇，上下界最小流 172](#_Toc466627802)

[在二分图中，用最少的点权，覆盖所有的边(二分图的最小点权覆盖集)。 173](#_Toc466627803)

[二分图的最大点权独立集算法 173](#_Toc466627804)

[费用与流量平方成正比 173](#_Toc466627805)

[刘汝佳书P367的流量不固定最小费用流 173](#_Toc466627806)

[公平分配问题 174](#_Toc466627807)

[区间K覆盖问题 174](#_Toc466627808)

[舞蹈链Dancing links 174](#_Toc466627809)

[简介 174](#_Toc466627810)

[不可重复覆盖代码板子（数独板子）来自kuangbin的板子 175](#_Toc466627811)

[附录 183](#_Toc466627812)

[线段树的几个常见板子 190](#_Toc466627813)

[一维线段树 190](#_Toc466627814)

[二维线段树 191](#_Toc466627815)

[有序表线段树 199](#_Toc466627816)

[附录 201](#_Toc466627817)

[斜率DP 207](#_Toc466627818)

[常见套路 207](#_Toc466627819)

[主席树 216](#_Toc466627820)

[线段树 216](#_Toc466627821)

[二、主席树 218](#_Toc466627822)

[莫比乌斯反演 223](#_Toc466627823)

[C++ pb\_ds库的 223](#_Toc466627824)

[几何模板： 228](#_Toc466627825)

[//##1##:点的定义 228](#_Toc466627826)

[//##2##:向量的基本运算 229](#_Toc466627827)

[//##3##:两直线交点（没有不能用）(P+tv & Q+tw，后面有Line, Line) 229](#_Toc466627828)

[//##4##:P点到直线AB的距离 230](#_Toc466627829)

[//##5##:P点到线段AB的距离 230](#_Toc466627830)

[//##6##:P点在直线AB上的投影 230](#_Toc466627831)

[//##7##:判断点P是否在线段a1a2上(不含端点) 230](#_Toc466627832)

[//##8##:线段a1a2,b1b2相交判定(线段无端点) 231](#_Toc466627833)

[//##9##:点的极角排序 231](#_Toc466627834)

[//##10##:有向直线。他的左边是对应的半平面 231](#_Toc466627835)

[//##11##:二直线交点，假定交点惟一存在 232](#_Toc466627836)

[//##12##:圆的定义 232](#_Toc466627837)

[//##13##:判断点在圆心内。圆周上不算 233](#_Toc466627838)

[//##14##:直线与圆的交点 233](#_Toc466627839)

[//##15##:圆和线段是否相交（相切不算）。线段不考虑端点 233](#_Toc466627840)

[//##16##:两圆相交的交点 234](#_Toc466627841)

[//##17##:把角变成0~2pi范围内(负数也可以用 234](#_Toc466627842)

[//##18##:两圆相交的交点相对于圆1的极角保存在rad中 234](#_Toc466627843)

[//##19##:过点p到圆C的切线。v[i]是第i条切线的向量。返回切线条数 235](#_Toc466627844)

[//##20##:过点p到圆C的切线。 235](#_Toc466627845)

[//##21##:过点p到圆C的切线 P为切点 236](#_Toc466627846)

[//##22##:两圆的公切线，返回切线条数，-1为无穷 236](#_Toc466627847)

[//##23##:三角形外接圆 237](#_Toc466627848)

[//##24##:三角形内切圆 237](#_Toc466627849)

[//##25##:经过点p且与直线L相切，半径为r的圆，返回值为圆心 237](#_Toc466627850)

[//##26##: 半径为r且与两条直线同时相切的圆 返回值为圆心 238](#_Toc466627851)

[//##27##: 求与两个圆外切的半径为r的圆, 返回圆心 238](#_Toc466627852)

[//##28##: 线段和圆的交点 结果保存在sol中 238](#_Toc466627853)

[//##29##: 多边形定义 239](#_Toc466627854)

[//##30##: 多边形的有向面积(点逆时针旋转为正向)(顺时针将求出负值) 239](#_Toc466627855)

[//##31##: 点在多边形内判定（不是简单多边形 有弧边也可以使用） 239](#_Toc466627856)

[//##32##: 多边形重心 240](#_Toc466627857)

[//##33##: 圆和凸多边形相交的面积 240](#_Toc466627858)

[//##34##: 返回圆盘是否与面poly相交 241](#_Toc466627859)

[//##35##: 删除平面的三点共线 241](#_Toc466627860)

[//##36##: 用有向直线AB切割poly 返回切割后的左边 242](#_Toc466627861)

[//##38##: 半平面交 243](#_Toc466627862)

[//##39##: 旋转卡壳求点集直径的平方 243](#_Toc466627863)

[//##40##: 旋转卡壳求能覆盖点集poly的 面积和周长最小的矩形 244](#_Toc466627864)

[//##41##: 平面直线图 245](#_Toc466627865)

[其他几何操作 247](#_Toc466627866)

[经纬度求球面最短距离 247](#_Toc466627867)

[三角形的心 247](#_Toc466627868)

[三角剖分 248](#_Toc466627869)

[凸包的相关操作 250](#_Toc466627870)

# 头文件

1. #include <bits/stdc++.h>
2. using std::sort;
3. using std::max;
4. using std::min;
5. using std::cout;
6. using std::cin;
7. using std::endl;
8. using std::swap;
9. using std::pair;
10. using std::map;
11. using std::vector;
12. using std::queue;
13. using std::string;
14. #define mp make\_pair
15. #define clr(x) memset(x,0,sizeof(x))
16. #define pr(x) cout<<#x<<" = "<<x<<" "
17. #define prln(x) cout<<#x<<" = "<<x<<endl
18. typedef unsigned long long uLL;
19. typedef long long LL;
20. typedef pair<int, int> pii;
21. typedef pair<double, double> pdd;
22. //const LL LINF = 0x3f3f3f3f3f3f3f3fll;//4e18
23. const int INF = 0x3f3f3f3f;//1e9
24. //不常用头文件
25. #include <ext/pb\_ds/priority\_queue.hpp>
26. #include <tr1/unordered\_map>
27. using std::tr1::unordered\_map;
28. using std::priority\_queue;
29. using std::set;
30. using std::make\_pair;
31. using std::unique;
32. using std::stack;
33. using std::multiset;
34. using std::greater;
35. using std::bitset;
36. using std::lower\_bound;//返回第一个不小于
37. using std::upper\_bound;//返回第一个大于
38. using std::max\_element;
39. using std::min\_element;
40. using std::for\_each;
41. using std::fill;
42. using std::\_\_gcd;
43. using \_\_gnu\_pbds::pairing\_heap\_tag;
44. #define x first
45. #define y second
46. #define Hash unordered\_map
47. #define logz(x) (31-\_\_builtin\_clz((int)x))
48. #define logzl(x) (63-\_\_builtin\_clzll((LL)x))
49. #define cf std::ios::sync\_with\_stdio(0);std::cin.tie(0)
50. #define lson o\*2, L, M
51. #define rson o\*2+1, M + 1,R
52. #define self o,L,R
53. typedef \_\_gnu\_pbds::priority\_queue<pii, greater<pii>, pairing\_heap\_tag> Heap;//小根堆
54. typedef Heap::point\_iterator Hit;
55. //const Hit null;
56. //const long double PI = acos(-1.0);
57. //const double eps = 1e-8;

# 快速读入

1. //比getchar快20倍
2. namespace IStream {
3. const int L = 1 << 15;
4. char buffer[L], \*S, \*T;
5. inline char get\_char() {
6. if (S == T) {
7. T = (S = buffer) + fread(buffer, 1, L, stdin);
8. if (S == T) return EOF;
9. }
10. return \*S++;
11. }
12. inline int get\_int(int& tvalue) {//只能读正整数，返回读入的数，读入失败返回0
13. char c;
14. int re = 0;
15. for (c = get\_char(); c<'0' || c>'9'; c = get\_char());
16. while (c >= '0' && c <= '9')
17. re=(re<<1)+(re<<3)+(c-'0'), c=get\_char();
18. return tvalue = re;
19. }
20. inline int get\_int2(int& tvalue) {//含有负数的整型读入, 成功返回1 失败返回EOF
21. char c;
22. int re = 0, sgn = 1;
23. for (c = get\_char(); c != EOF && c != '-' && (c<'0' || c>'9'); c = get\_char());
24. if (c == EOF) return EOF;
25. if (c == '-') { sgn = -1; c = get\_char(); }
26. while (c >= '0' && c <= '9')
27. re=(re<<1)+(re<<3)+(c-'0'), c=get\_char();
28. tvalue = sgn \* re;
29. return 1;
30. }
31. inline int get\_double(double& tvalue) {//含有负数的double读入, 成功返回1 失败返回EOF
32. char c;
33. int re = 0, sgn = 1;
34. double f = 0, base = 0.1;
35. for (c = get\_char(); c != EOF && c != '-' && (c<'0' || c>'9'); c = get\_char());
36. if (c == EOF) return EOF;
37. if (c == '-') { sgn = -1; c = get\_char(); }
38. while (c >= '0' && c <= '9')
39. re=(re<<1)+(re<<3)+(c-'0'), c=get\_char();
40. if (c == '.') {
41. c = get\_char();
42. while (c >= '0' && c <= '9') {
43. f += base \* (c-'0');
44. base /= 10;
45. c = get\_char();
46. }
47. }
48. tvalue = sgn \* re + f;
49. return 1;
50. }
51. //和普通gets一样快
52. bool get\_s(char\* s) { //成功返回true，失败返回false
53. char c = get\_char();
54. while (c == '\n') c = get\_char(); //读到第一个不是回车符
55. if (c == EOF) return false;
56. while (c != '\n' && c != EOF) {
57. \*s++ = c;
58. c = get\_char();
59. }
60. \*s = 0;
61. return true;
62. }
63. }
64. using IStream::get\_int2;
65. using IStream::get\_double;
66. using IStream::get\_s;

# 快速输出

1. //比printf快20倍
2. //调用方法 print\_int(LL num) 可以输出LL
3. //write\_s(char\* s, int len = -1); 可以输出任意字符串s
4. //warning:
5. //return 0;之前 一定要调用一次clear\_write();
6. //任意字符串s的长度不能超过MAX\_LEN
7. namespace OStream {
8. const int BUFSIZE = 1 << 14;//缓冲区的最大字符数量
9. const int MAX\_LEN = 1 << 10;//1024
10. static char outbuf[BUFSIZE + MAX\_LEN];
11. static char \*outp = outbuf;
12. static char \*loutp = outbuf + BUFSIZE;
13. inline void clear\_write() {
14. cout.write(outbuf, outp - outbuf);
15. }
16. inline void write\_s(char\* s, int len = -1) {
17. if (len == -1) len = strlen(s);
18. for (int i = 0; i < len; i++) {
19. \*outp++ = s[i];
20. }
21. if (outp > loutp) {//超过BUSFIZE个元素的时候才输出
22. cout.write(outbuf, outp - outbuf);
23. outp = outbuf;
24. }
25. }
26. char t[20], s[20];
27. inline void print\_int(LL num)
28. {
29. int len = (num == 0);
30. t[0] = '0';
31. while (num) {
32. t[len++] = (num % 10) + '0';
33. num /= 10;
34. }
35. for (int i = 0; i < len; i++) s[i] = t[len-i-1];
36. s[len] = 0;
37. write\_s(s, len);
38. }
39. }
40. using OStream::clear\_write;
41. using OStream::write\_s;
42. using OStream::print\_int;

# ---------图论：

## 弦图判断

1. /\*==================================================\*\
2. | 弦图判断
3. | 弦图：图中任意的大于3的环 都至少有一个弦
4. | INIT: g[][]置为邻接矩阵;
5. | CALL: mcs(n); peo(n);
6. | 第一步: 给节点编号 mcs(n)
7. | 设已编号的节点集合为A, 未编号的节点集合为B
8. | 开始时A为空, B包含所有节点.
9. | for num=n-1 downto 0 do {
10. | 在B中找节点x, 使与x相邻的在A集合中的节点数最多,
11. | 将x编号为num, 并从B移入A.
12. | }
13. | 第二步: 检查 peo(n)
14. | for num=0 to n-1 do {
15. | 对编号为num的点x, 设所有编号>num且与x相邻的点集为C
16. | 在C中找出编号最小的节点y,
17. | 若C中存在点z!=y, 使得y与z之间无边, 则此图不是弦图.
18. | }
19. | 检查完了, 则此图是弦图.
20. \\*==================================================\*/
21. const int V = 100;
22. int g[V][V], order[V], inv[V], tag[V];
23. void mcs(int n){
24. int i, j, k;
25. memset(tag, 0, sizeof(tag));
26. memset(order, -1, sizeof(order));
27. for (i = n - 1; i >= 0; i--) { // vertex: 0 ~ n-1
28. for (j = 0; order[j] >= 0; j++) ;
29. for (k = j + 1; k < n; k++)
30. if (order[k] < 0 && tag[k] > tag[j]) j = k;
31. order[j] = i, inv[i] = j;
32. for (k = 0; k < n; k++) if (g[j][k]) tag[k]++;
33. }
34. }
35. int peo(int n){
36. int i, j, k, w, min;
37. for (i = n - 2; i >= 0; i--) {
38. j = inv[i], w = -1, min = n;
39. for (k = 0; k < n; k++)
40. if (g[j][k] && order[k] > order[j] &&
41. order[k] < min)
42. min = order[k], w=k;
43. if (w < 0) continue;
44. for (k = 0; k < n; k++)
45. if (g[j][k] && order[k] > order[w] && !g[k][w])
46. return 0; // no
47. }
48. return 1; // yes
49. }

## 快速最大团

1. //快速最大团
2. // Super Fast Maximum Clique
3. // To Build Graph: Maxclique(Edges, Number of Nodes)
4. // To Get Answer: mcqdyn(AnswerNodes Index Array, AnswserLength)
5. typedef bool BB[N];
6. struct Maxclique {
7. const BB\* e; int pk, level; const float Tlimit;
8. struct Vertex{ int i, d; Vertex(int i):i(i),d(0){} };
9. typedef vector<Vertex> Vertices; typedef vector<int> ColorClass;
10. Vertices V; vector<ColorClass> C; ColorClass QMAX, Q;
11. static bool desc\_degree(const Vertex &vi, const Vertex &vj){
12. return vi.d > vj.d;
13. }
14. void init\_colors(Vertices &v){
15. const int max\_degree = v[0].d;
16. for(int i = 0; i < (int)v.size(); i++) v[i].d = min(i, max\_degree) + 1;
17. }
18. void set\_degrees(Vertices &v){
19. for(int i = 0, j; i < (int)v.size(); i++)
20. for(v[i].d = j = 0; j < int(v.size()); j++)
21. v[i].d += e[v[i].i][v[j].i];
22. }
23. struct StepCount{ int i1, i2; StepCount():i1(0),i2(0){} };
24. vector<StepCount> S;
25. bool cut1(const int pi, const ColorClass &A){
26. for(int i = 0; i < (int)A.size(); i++) if (e[pi][A[i]]) return true;
27. return false;
28. }
29. void cut2(const Vertices &A, Vertices &B){
30. for(int i = 0; i < (int)A.size() - 1; i++)
31. if(e[A.back().i][A[i].i])
32. B.push\_back(A[i].i);
33. }
34. void color\_sort(Vertices &R){
35. int j = 0, maxno = 1, min\_k = max((int)QMAX.size() - (int)Q.size() + 1, 1);
36. C[1].clear(), C[2].clear();
37. for(int i = 0; i < (int)R.size(); i++) {
38. int pi = R[i].i, k = 1;
39. while(cut1(pi, C[k])) k++;
40. if(k > maxno) maxno = k, C[maxno + 1].clear();
41. C[k].push\_back(pi);
42. if(k < min\_k) R[j++].i = pi;
43. }
44. if(j > 0) R[j - 1].d = 0;
45. for(int k = min\_k; k <= maxno; k++)
46. for(int i = 0; i < (int)C[k].size(); i++)
47. R[j].i = C[k][i], R[j++].d = k;
48. }
49. void expand\_dyn(Vertices &R){// diff -> diff with no dyn
50. S[level].i1 = S[level].i1 + S[level - 1].i1 - S[level].i2;//diff
51. S[level].i2 = S[level - 1].i1;//diff
52. while((int)R.size()) {
53. if((int)Q.size() + R.back().d > (int)QMAX.size()){
54. Q.push\_back(R.back().i); Vertices Rp; cut2(R, Rp);
55. if((int)Rp.size()){
56. if((float)S[level].i1 / ++pk < Tlimit) degree\_sort(Rp);//diff
57. color\_sort(Rp);
58. S[level].i1++, level++;//diff
59. expand\_dyn(Rp);
60. level--;//diff
61. }
62. else if((int)Q.size() > (int)QMAX.size()) QMAX = Q;
63. Q.pop\_back();
64. }
65. else return;
66. R.pop\_back();
67. }
68. }
69. void mcqdyn(int\* maxclique, int &sz){
70. set\_degrees(V); sort(V.begin(),V.end(), desc\_degree); init\_colors(V);
71. for(int i = 0; i < (int)V.size() + 1; i++) S[i].i1 = S[i].i2 = 0;
72. expand\_dyn(V); sz = (int)QMAX.size();
73. for(int i = 0; i < (int)QMAX.size(); i++) maxclique[i] = QMAX[i];
74. }
75. void degree\_sort(Vertices &R){
76. set\_degrees(R); sort(R.begin(), R.end(), desc\_degree);
77. }
78. Maxclique(const BB\* conn, const int sz, const float tt = 0.025) \
79. : pk(0), level(1), Tlimit(tt) {
80. for(int i = 0; i < sz; i++) V.push\_back(Vertex(i));
81. e = conn, C.resize(sz + 1), S.resize(sz + 1);
82. }
83. };

## 求最大团的点数

1. // 求最大团的点数
2. //warning: 下标从1开始 mc[1] 就是最大团的点数
3. //int g[][] 为图的邻接矩阵 标号由 1 至 n
4. //MC(V) 表示点集 V 的最大团 令 Si={vi, vi+1, ..., vn}, mc[i] 表示 MC(Si)
5. //倒着算 mc[i], 那么显然 MC(V)=mc[1] 此外有 mc[i]=mc[i+1] or mc[i]=mc[i+1]+1
6. int n; //总点数
7. bool found;
8. int len[100], int mc[100], int list[100][100];
9. int ans;
10. void dfs(int size){
11. if (len[size]==0) {
12. if (size>ans) ans=size, found=true; return;
13. } for (int k=0,i,j; k<len[size] && !found; ++k) {
14. if (size+len[size]-k<=ans) break;
15. i=list[size][k]; if (size+mc[i]<=ans) break;
16. for (j=k+1, len[size+1]=0; j<len[size]; ++j)
17. if (g[i][list[size][j]]) list[size+1][len[size+1]++]=list[size][j];
18. dfs(size+1);
19. }}
20. void work(){
21. mc[n]=ans=1;
22. for (int i=n-1; i; --i) {
23. found=false; len[1]=0;
24. for (int j=i+1; j<=n; ++j) if (g[i][j]) list[1][len[1]++]=j;
25. dfs(1); mc[i]=ans;
26. }}

## 二分图匹配的匈牙利算法

1. vector<int> G[maxn];
2. bool vis[maxn];
3. int match[maxn];
4. bool dfs(int u)
5. {
6. for (int i = 0; i < G[u].size(); i++)
7. {
8. int v = G[u][i];
9. if (vis[v]) continue;
10. vis[v] = true;
11. if (match[v] == -1 || dfs(match[v]))
12. {
13. match[v] = u;
14. return true;
15. }
16. }
17. return false;
18. }
19. int hungary(int n) //传入二分图一边的节点数
20. {
21. int matches = 0;
22. memset(match, -1, sizeof match);
23. for(int i = 1; i <= n; ++i)
24. {
25. memset(vis, 0, sizeof vis);
26. matches += dfs(i);
27. }
28. return matches;
29. }

## 基于spfa的最小费用流

1. //调用方法：
2. //init(n) 初始化 n 为节点数
3. //add\_edge(from, to, cap, cost) 添加边from -> to 容量为cap 费用为cost
4. //（自动添加反向边(容量为0，费用为-cost）
5. //int mincost\_maxflow(int s, int t, LL& cost) //需要保证初始网络中没有负圈 最小费用最大流
6. //如果只是找最小费用（不在意流量）开启bellman\_ford里面的一个注释，并注释下一行
7. struct Edge {
8. int from, to, cap, flow, cost;
9. };
10. struct MCMF {
11. int n, m, s, t;
12. vector<Edge> edges;
13. vector<int> G[maxn];
14. bool inq[maxn]; // 是否在队列中
15. int d[maxn]; // Bellman-Ford
16. int p[maxn]; // 上一条弧
17. int a[maxn]; // 可改进量
18. void init(int n) {
19. this->n = n;
20. for (int i = 0; i < n; i++) G[i].clear();
21. edges.clear();
22. }
23. void add\_edge(int from, int to, int cap, int cost) {
24. edges.push\_back((Edge){from, to, cap, 0, cost});
25. edges.push\_back((Edge){to, from, 0, 0, -cost});
26. m = edges.size();
27. G[from].push\_back(m-2);
28. G[to].push\_back(m-1);
29. }
30. bool bellman\_ford(int s, int t, int& flow, LL& cost) {
31. for (int i = 0; i < n; i++) d[i] = INF;//初始化 每个点到起点的费用
32. memset(inq, 0, sizeof(inq));//每个点都没有入队
33. memset(cnt, 0, sizeof(cnt));
34. d[s] = 0; inq[s] = true; p[s] = 0; a[s] = INF;//起点的状态
35. queue<int> q;
36. q.push(s);
37. while (!q.empty()) {
38. int u = q.front(); q.pop();
39. inq[u] = false;
40. for (int i = 0; i < G[u].size(); i++) {
41. Edge& e = edges[G[u][i]];
42. if (e.cap > e.flow && d[e.to] > d[u] + e.cost) {
43. d[e.to] = d[u] + e.cost;
44. p[e.to] = G[u][i];
45. a[e.to] = min(a[u], e.cap - e.flow);
46. if (!inq[e.to]) {
47. q.push(e.to); inq[e.to] = true;
48. }
49. }
50. }
51. }
52. // if (d[t] > 0) return false; //找最小费用，不管流量（例如收益等）开启，注释下一行
53. if (d[t] == INF) return false;
54. flow += a[t];
55. cost += (LL)d[t] \* (LL)a[t];
56. for (int u = t; u != s; u = edges[p[u]].from) {
57. edges[p[u]].flow += a[t];
58. edges[p[u]^1].flow -= a[t];
59. }
60. return true;
61. }
62. //需要保证初始网络中没有负圈 最小费用最大流
63. int mincost\_maxflow(int s, int t, LL& cost) {
64. int flow = 0; cost = 0;
65. while (bellman\_ford(s, t, flow, cost)); //走不通才退出
66. return flow;
67. }
68. } g;

## ISAP 求最大流 最小割

1. //调用方法：
2. //init(n) 初始化 n 为节点数
3. //clear\_flow() 清空所有边的流量
4. //add\_edge(from, to, cap) 添加边from -> to 容量为cap
5. //maxflow(s, t) 返回s->t的最大流
6. //void mincut(vector<int>& ans) // 调用完maxflow后才可以用，ans里面存最小割
7. //print() 打印整张图，调试用
8. struct Edge { int from, to, cap, flow; };
9. struct ISAP {
10. int n, m, s, t;
11. vector<Edge> edges;
12. vector<int> G[maxn]; // 邻接表，G[i][j]表示结点i的第j条边在e数组中的序号
13. bool vis[maxn]; // BFS使用
14. int d[maxn]; // 从起点到i的距离
15. int cur[maxn]; // 当前弧指针
16. int p[maxn]; // 可增广路上的上一条弧
17. int num[maxn]; // 距离标号计数
18. void add\_edge(int from, int to, int cap) {
19. edges.push\_back((Edge){from, to, cap, 0});
20. edges.push\_back((Edge){to, from, 0, 0});
21. m = edges.size();
22. G[from].push\_back(m-2);
23. G[to].push\_back(m-1);
24. }
25. bool bfs() {
26. memset(vis, 0, sizeof(vis));
27. queue<int> q;
28. q.push(t);
29. vis[t] = 1;
30. d[t] = 0;
31. while(!q.empty()) {
32. int x = q.front(); q.pop();
33. for(int i = 0; i < G[x].size(); i++) {
34. Edge& e = edges[G[x][i]^1];
35. if(!vis[e.from] && e.cap > e.flow) {
36. vis[e.from] = 1;
37. d[e.from] = d[x] + 1;
38. q.push(e.from);
39. }
40. }
41. }
42. return vis[s];
43. }
44. void init(int n) {
45. this->n = n;
46. for(int i = 0; i < n; i++) G[i].clear();
47. edges.clear();
48. }
49. void clear\_flow() {
50. for(int i = 0; i < edges.size(); i++) edges[i].flow = 0;
51. }
52. int augment() {
53. int x = t, a = INF;
54. while(x != s) {
55. Edge& e = edges[p[x]];
56. a = min(a, e.cap-e.flow);
57. x = edges[p[x]].from;
58. }
59. x = t;
60. while(x != s) {
61. edges[p[x]].flow += a;
62. edges[p[x]^1].flow -= a;
63. x = edges[p[x]].from;
64. }
65. return a;
66. }
67. int maxflow(int s, int t, int need) {//找到的最大流大于need就停止，如果没有限制，删去含有need的地方
68. this->s = s; this->t = t;
69. int flow = 0;
70. bfs();
71. memset(num, 0, sizeof(num));
72. for(int i = 0; i < n; i++) num[d[i]]++;
73. int x = s;
74. memset(cur, 0, sizeof(cur));
75. while(d[s] < n) {
76. if(x == t) {
77. flow += augment();
78. if(flow >= need) return flow;
79. x = s;
80. }
81. int ok = 0;
82. for(int i = cur[x]; i < G[x].size(); i++) {
83. Edge& e = edges[G[x][i]];
84. if(e.cap > e.flow && d[x] == d[e.to] + 1) { // Advance
85. ok = 1;
86. p[e.to] = G[x][i];
87. cur[x] = i; // 注意
88. x = e.to;
89. break;
90. }
91. }
92. if(!ok) { // Retreat
93. int m = n-1; // 初值注意
94. for(int i = 0; i < G[x].size(); i++) {
95. Edge& e = edges[G[x][i]];
96. if(e.cap > e.flow) m = min(m, d[e.to]);
97. }
98. if(--num[d[x]] == 0) break;//gap优化
99. num[d[x] = m+1]++;
100. cur[x] = 0; // 注意
101. if(x != s) x = edges[p[x]].from;
102. }
103. }
104. return flow;
105. }
106. void mincut(vector<int>& ans) { // 调用完maxflow后才可以用，ans里面存最小割
107. bfs();
108. for(int i = 0; i < edges.size(); i++) {
109. Edge& e = edges[i];
110. if(!vis[e.from] && vis[e.to] && e.cap > 0) ans.push\_back(i);
111. }
112. }
113. void print() {
114. printf("Graph:\n");
115. for(int i = 0; i < edges.size(); i++)
116. printf("%d->%d, %d, %d\n", edges[i].from, edges[i].to , edges[i].cap, edges[i].flow);
117. }
118. } isap;

## Dinic 求最大流 最小割

复杂度:

如果所有容量均为1 复杂度O(min(n^(2/3),  m^(1/2))m)

除了源点和汇点之外，每个点要么只有一条入弧，且容量为1，要么只有一条出弧，且容量为1，其他弧容量为任意整数

时间复杂度为O(n^(1/2)m)

1. //调用方法：
2. //init(n) 初始化 n 为节点数
3. //clear\_nodes(a, b) 清空节点a 到 节点b的边
4. //clear\_flow() 清空所有边的流量
5. //add\_edge(from, to, cap) 添加边from -> to 容量为cap
6. //maxflow(s, t) 返回s->t的最大流
7. //int maxflow\_under\_limit(int s, int t, int limit)
8. //求s-t最大流。如果最大流超过limit，则只找一个流量为limit的流
9. //void mincut(vector<int>& ans) // 调用完maxflow后才可以用，ans里面存最小割
10. struct Edge { int from, to, cap, flow; };
11. struct Dinic {
12. int n, m, s, t; //这里的n好像没有作用
13. vector<Edge> edges; // 边数的两倍
14. vector<int> G[maxn]; // 邻接表，G[i][j]表示结点i的第j条边在e数组中的序号
15. bool vis[maxn]; // bfs使用
16. int d[maxn]; // 从起点到i的距离
17. int cur[maxn]; // 当前弧指针
18. void init(int n) {
19. for(int i = 0; i < n; i++) G[i].clear();
20. edges.clear();
21. }
22. void clear\_nodes(int a, int b) {//清空节点a到b的边
23. for (int i = a; i <= b; i++) G[i].clear();
24. }
25. void clear\_flow() {
26. for(int i = 0; i < edges.size(); i++) edges[i].flow = 0;
27. }
28. void add\_edge(int from, int to, int cap) {
29. edges.push\_back((Edge){from, to, cap, 0});
30. edges.push\_back((Edge){to, from, 0, 0});
31. m = edges.size();
32. G[from].push\_back(m-2);
33. G[to].push\_back(m-1);
34. }
35. bool bfs() {
36. memset(vis, 0, sizeof(vis));
37. queue<int> q;
38. q.push(s);
39. vis[s] = 1;
40. d[s] = 0;
41. while(!q.empty()) {
42. int x = q.front(); q.pop();
43. for(int i = 0; i < G[x].size(); i++) {
44. Edge& e = edges[G[x][i]];
45. if(!vis[e.to] && e.cap > e.flow) {
46. vis[e.to] = 1;
47. d[e.to] = d[x] + 1;
48. q.push(e.to);
49. }
50. }
51. }
52. return vis[t];
53. }
54. int dfs(int x, int a) {//到目前为止所有弧中最小的残量
55. if(x == t || a == 0) return a;
56. int flow = 0, f;
57. for(int& i = cur[x]; i < G[x].size(); i++) {
58. Edge& e = edges[G[x][i]];
59. if(d[x] + 1 == d[e.to] && (f = dfs(e.to, min(a, e.cap-e.flow))) > 0) {
60. e.flow += f;
61. edges[G[x][i]^1].flow -= f;
62. flow += f;
63. a -= f;
64. if(a == 0) break;
65. }
66. }
67. return flow;
68. }
69. int maxflow(int s, int t) {
70. this->s = s; this->t = t;
71. int flow = 0;
72. while(bfs()) {
73. memset(cur, 0, sizeof(cur));
74. flow += dfs(s, INF);
75. }
76. return flow;
77. }
78. // 求s-t最大流。如果最大流超过limit，则只找一个流量为limit的流
79. int maxflow\_under\_limit(int s, int t, int limit) {
80. this->s = s; this->t = t;
81. int flow = 0;
82. while(bfs()) {
83. memset(cur, 0, sizeof(cur));
84. flow += dfs(s, limit - flow);
85. if(flow == limit) break; // 达到流量限制，直接退出
86. }
87. return flow;
88. }
89. void mincut(vector<int>& ans) { // 调用完maxflow后才可以用，ans里面存最小割
90. for(int i = 0; i < edges.size(); i++) {
91. Edge& e = edges[i];
92. if(vis[e.from] && !vis[e.to] && e.cap > 0) ans.push\_back(i);
93. }
94. }
95. } din;

## 求最大匹配、最小覆盖

1. //点-最小覆盖=最大独立集
2. //DAG最小路径覆盖的解法：
3. //把所有结点i拆成x结点i和y结点i'，如果图G中存在有向边i->j，则在二分图中引入边i->j'。
4. //结果就是原点数-二分图的最大匹配数
5. const int maxn = 1000 + 5; // 单侧顶点的最大数目
6. // 二分图最大基数匹配
7. //
8. // 调用方法：
9. // init(n, m) 初始化左右点个数
10. // add\_edge(u, v) 添边u->v
11. // solve() 求最大匹配
12. //int mincover(vector<int>& X, vector<int>& Y) 求最小覆盖。X和Y为最小覆盖中的点集
13. //返回值为最大匹配数
14. struct BPM {
15. int n, m; // 左右顶点个数
16. vector<int> G[maxn]; // 邻接表
17. int left[maxn]; // left[i]为右边第i个点的匹配点编号，-1表示不存在
18. bool T[maxn]; // T[i]为右边第i个点是否已标记
19. int right[maxn]; // 求最小覆盖用
20. bool S[maxn]; // 求最小覆盖用
21. void init(int n, int m) {
22. this->n = n;
23. this->m = m;
24. for(int i = 0; i < n; i++) G[i].clear();
25. }
26. void add\_edge(int u, int v) {
27. G[u].push\_back(v);
28. }
29. bool match(int u){
30. S[u] = true;
31. for(int i = 0; i < G[u].size(); i++) {
32. int v = G[u][i];
33. if (!T[v]) {
34. T[v] = true;
35. if (left[v] == -1 || match(left[v])) {
36. left[v] = u;
37. right[u] = v;
38. return true;
39. }
40. }
41. }
42. return false;
43. }
44. // 求最大匹配
45. int solve() {
46. memset(left, -1, sizeof(left));
47. memset(right, -1, sizeof(right));
48. int ans = 0;
49. for(int u = 0; u < n; u++) { // 从左边结点u开始增广
50. memset(S, 0, sizeof(S));
51. memset(T, 0, sizeof(T));
52. if(match(u)) ans++;
53. }
54. return ans;
55. }
56. // 求最小覆盖。X和Y为最小覆盖中的点集
57. int mincover(vector<int>& X, vector<int>& Y) {
58. int ans = solve();
59. memset(S, 0, sizeof(S));
60. memset(T, 0, sizeof(T));
61. for(int u = 0; u < n; u++)
62. if(right[u] == -1) match(u); // 从所有X未盖点出发增广
63. for(int u = 0; u < n; u++)
64. if(!S[u]) X.push\_back(u); // X中的未标记点
65. for(int v = 0; v < m; v++)
66. if(T[v]) Y.push\_back(v); // Y中的已标记点
67. return ans;//返回值为最大匹配数
68. }
69. } bpm;

## 稳定婚姻问题

输入

1、男生按喜欢程度输入每个女生的编号

2、女生按喜欢程度输入每个男生的编号

输出

稳定的匹配方案：每个男生在一起的女生的编号

复杂度:O(n2)

测试数据：

输入：

1 5

1 2 3 5 4 5 2 4 3 1 3 5 1 2 4 3 4 2 1 5 4 5 1 2 3 2 5 4 1 3 3 2 4 1 5 1 2 4 3 5 4 1 2 5 3 5 3 2 4 1

输出：1\n2\n5\n3\n4

1. const int maxn = 1000 + 10;
2. int pref[maxn][maxn], order[maxn][maxn], next[maxn], future\_husband[maxn], future\_wife[maxn];
3. queue<int> q; // 未订婚的男士队列
4. int n;
5. void init()
6. {
7. scanf("%d", &n);
8. for (int i = 1; i <= n; i++)
9. {
10. for (int j = 1; j <= n; j++)
11. {
12. scanf("%d", &pref[i][j]);//编号为i的男士 第j喜欢的人
13. }
14. next[i] = 1;//初始化下一个求婚对象（即第一个喜欢的人
15. future\_wife[i] = 0;//i没有未婚妻
16. q.push(i);
17. }
18. for (int i = 1; i <= n; i++)
19. {
20. for (int j = 1; j <= n; j++)
21. {
22. int x;
23. scanf("%d", &x);
24. order[i][x] = j;//在编号为i的女士的心目中，编号为x的男士的排名
25. }
26. future\_husband[i] = 0;//没有未婚夫
27. }
28. }
29. void engage(int man, int woman)
30. {
31. int m = future\_husband[woman];
32. if (m) {//妹子如果有未婚夫m, 抛弃他
33. future\_wife[m] = 0;
34. q.push(m);
35. }
36. //和更喜欢的结合
37. future\_wife[man] = woman;
38. future\_husband[woman] = man;
39. }
40. void solve()
41. {
42. while (!q.empty())
43. {
44. int man = q.front(); q.pop();
45. int woman = pref[man][next[man]++];//编号为i的男士下一个喜欢的人
46. if (!future\_husband[woman]) engage(man, woman);//如果这个妹子也没有未婚夫 就订婚
47. else if (order[woman][man] < order[woman][future\_husband[woman]])
48. engage(man, woman);//如果这个妹子相对于她的未婚夫更喜欢现在的男的，就和他订婚
49. else q.push(man); //没人要就走下一轮吧
50. }
51. for (int i = 1; i <= n; i++) printf("%d\n", future\_wife[i]); //输出每个男士的未婚妻
52. }
53. int main()
54. {
55. int T;
56. scanf("%d", &T);
57. while (T--)
58. {
59. init();
60. solve();
61. if (T) printf("\n");
62. }
63. return 0;
64. }

## 求二分图最佳完美匹配的KM算法

1. // 最大权匹配（下标从0开始
2. // 左右顶点均从0编号
3. // init(n) 传入左右点数的最大值
4. // add\_edge(u, v, w) 添加边u->v 权值为w
5. // 执行solve() 得到left, left[i]为右边第i个点的匹配点编号，-1表示不存在
6. // Lx[i] + Ly[j] >= W[i][j];
7. struct KM {
8. int n; // 左右顶点个数(左右顶点的最大值，不存在的点补全，并且添上没用的边，放在后面都用一个数
9. vector<int> G[maxn]; // 邻接表
10. int W[maxn][maxn]; // 权值
11. int Lx[maxn], Ly[maxn]; // 顶标（执行完后所有顶标之和最小
12. int left[maxn]; // left[i]为右边第i个点的匹配点编号，-1表示不存在
13. bool S[maxn], T[maxn]; // S[i]和T[i]为左/右第i个点是否已标记
14. int slack[maxn]; // 松弛量 slack[y] = min(Lx[x] + Ly[y] - W[x][y])
15. void init(int n) {
16. this->n = n;
17. for (int i = 0; i < n; i++) G[i].clear();
18. memset(W, 0, sizeof(W));
19. }
20. void add\_edge(int u, int v, int w) {
21. G[u].push\_back(v);
22. W[u][v] = w;
23. }
24. bool match(int u) {
25. S[u] = true;
26. for (int i = 0; i < G[u].size(); i++) {
27. int v = G[u][i];
28. if (Lx[u] + Ly[v] == W[u][v] && !T[v]) {
29. T[v] = true;
30. if (left[v] == -1 || match(left[v])) {
31. left[v] = u;
32. return true;
33. }
34. }
35. else slack[v] = min(slack[v], Lx[u] + Ly[v] - W[u][v]);
36. }
37. return false;
38. }
39. void update() {
40. int a = INF;
41. for (int i = 0; i < n; i++) if (!T[i]) a = min(a, slack[i]);
42. for (int i = 0; i < n; i++) {
43. if (S[i]) Lx[i] -= a;
44. if (T[i]) Ly[i] += a; else slack[i] -= a;
45. }
46. }
47. void solve() {
48. for (int i = 0; i < n; i++) {
49. Lx[i] = \*max\_element(W[i], W[i]+n);
50. left[i] = -1;
51. Ly[i] = 0;
52. }
53. for (int u = 0; u < n; u++) {
54. memset(slack, 0x3f, sizeof(slack));
55. for (;;) {
56. memset(S, 0, sizeof(S));
57. memset(T, 0, sizeof(T));
58. if (match(u)) break;
59. update();//如果相等子图没有完美匹配，就更新顶标
60. }
61. }
62. }
63. } km;

## 最小树形图

调用方法：

init节点个数

add\_edge添加边

就可以跑了 has\_mdst

1. // 固定根的最小树型图，邻接矩阵写法
2. struct MDST {
3. int n;
4. int w[maxn][maxn]; // 边权
5. int vis[maxn]; // 访问标记，仅用来判断无解
6. int ans; // 计算答案
7. int removed[maxn]; // 每个点是否被删除
8. int cid[maxn]; // 所在圈编号
9. int pre[maxn]; // 最小入边的起点
10. int iw[maxn]; // 最小入边的权值
11. int max\_cid; // 最大圈编号
12. void init(int n) {
13. this->n = n;
14. for(int i = 0; i < n; i++)
15. for(int j = 0; j < n; j++) w[i][j] = INF;
16. }
17. void add\_edge(int u, int v, int cost) {
18. w[u][v] = min(w[u][v], cost); // 重边取权最小的
19. }
20. // 从s出发能到达多少个结点
21. int dfs(int s) {
22. vis[s] = 1;
23. int ans = 1;
24. for(int i = 0; i < n; i++)
25. if(!vis[i] && w[s][i] < INF) ans += dfs(i);
26. return ans;
27. }
28. // 从u出发沿着pre指针找圈
29. bool cycle(int u) {
30. max\_cid++;
31. int v = u;
32. while(cid[v] != max\_cid) { cid[v] = max\_cid; v = pre[v]; }
33. return v == u;
34. }
35. // 计算u的最小入弧，入弧起点不得在圈c中
36. void update(int u) {
37. iw[u] = INF;
38. for(int i = 0; i < n; i++)
39. if(!removed[i] && w[i][u] < iw[u]) {
40. iw[u] = w[i][u];
41. pre[u] = i;
42. }
43. }
44. // 根结点为s，如果失败则返回false
45. bool has\_mdst(int s) {
46. memset(vis, 0, sizeof(vis));
47. if(dfs(s) != n) return false;
48. memset(removed, 0, sizeof(removed));
49. memset(cid, 0, sizeof(cid));
50. for(int u = 0; u < n; u++) update(u);
51. pre[s] = s; iw[s] = 0; // 根结点特殊处理
52. ans = max\_cid = 0;
53. for(;;) {
54. bool have\_cycle = false;
55. for(int u = 0; u < n; u++) if(u != s && !removed[u] && cycle(u)){
56. have\_cycle = true;
57. // 以下代码缩圈，圈上除了u之外的结点均删除
58. int v = u;
59. do {
60. if(v != u) removed[v] = 1;
61. ans += iw[v];
62. // 对于圈外点i，把边i->v改成i->u（并调整权值）；v->i改为u->i
63. // 注意圈上可能还有一个v'使得i->v'或者v'->i存在，因此只保留权值最小的i->u和u->i
64. for(int i = 0; i < n; i++) if(cid[i] != cid[u] && !removed[i]) {
65. if(w[i][v] < INF) w[i][u] = min(w[i][u], w[i][v]-iw[v]);
66. w[u][i] = min(w[u][i], w[v][i]);
67. if(pre[i] == v) pre[i] = u;
68. }
69. v = pre[v];
70. } while(v != u);
71. update(u);
72. break;
73. }
74. if(!have\_cycle) break;
75. }
76. for(int i = 0; i < n; i++)
77. if(!removed[i]) ans += iw[i];
78. return true;
79. }
80. } md;

## 最近公共祖先

1. //调用方法：
2. //初始化init(n, fa, cost, L)
3. //分别传入总点数，父亲数组，节点和其父亲的费用，所在的层次（根为0）
4. //preprocess() 预处理，根据fa和cost数组求出anc和maxcost数组
5. //anc[p][i]是结点p的第2^i级父亲。anc[i][0] = fa[i]
6. //maxcost[p][i]是i和anc[p][i]的路径上的最大费用
7. //query(int p, int q) 求p到q的路径上的最大权
8. //(或者求p q的最早公共祖先，具体修改方法见代码注释)
9. struct LCA {
10. int n;
11. int\* fa; // 父亲数组
12. int\* cost; // 和父亲的费用
13. int\* L; // 层次（根节点层次为0）
14. int anc[maxn][maxlog]; // anc[p][i]是结点p的第2^i级父亲。anc[i][0] = fa[i]
15. int maxcost[maxn][maxlog]; // maxcost[p][i]是i和anc[p][i]的路径上的最大费用
16. void init(int n, int\* fa, int\* cost, int\* L) {
17. this->n = n;
18. this->fa = fa;
19. this->cost = cost;
20. this->L = L;
21. }
22. // 预处理，根据fa和cost数组求出anc和maxcost数组
23. void preprocess() {
24. for(int i = 0; i < n; i++) {
25. anc[i][0] = fa[i]; maxcost[i][0] = cost[i];
26. for(int j = 1; (1 << j) < n; j++) anc[i][j] = -1;
27. }
28. for(int j = 1; (1 << j) < n; j++) {
29. for(int i = 0; i < n; i++) {
30. if(anc[i][j-1] != -1) {
31. int a = anc[i][j-1];
32. anc[i][j] = anc[a][j-1];
33. maxcost[i][j] = max(maxcost[i][j-1], maxcost[a][j-1]);
34. }
35. }
36. }
37. }
38. // 求p到q的路径上的最大权
39. int query(int p, int q) {
40. int tmp, power, i;
41. if(L[p] < L[q]) swap(p, q); //L[p] >= L[q]
42. for(power = 1; (1 << power) <= L[p]; power++);
43. power--; //(2^power <= L[p]中的最大的)
44. int ans = -INF;
45. for(int i = power; i >= 0; i--) {
46. if (L[p] - (1 << i) >= L[q]) {
47. ans = max(ans, maxcost[p][i]);
48. p = anc[p][i];
49. }
50. }
51. if (p == q) return ans; // LCA为p
52. for(int i = power; i >= 0; i--) {
53. if(anc[p][i] != -1 && anc[p][i] != anc[q][i]) {
54. ans = max(ans, maxcost[p][i]); p = anc[p][i];
55. ans = max(ans, maxcost[q][i]); q = anc[q][i];
56. }
57. }
58. ans = max(ans, cost[p]);
59. ans = max(ans, cost[q]);
60. return ans; // LCA为fa[p]（它也等于fa[q]）
61. }
62. } lca;

## 2-SAT

1. //调用方法：
2. //G为邻接表 下标从0开始
3. //先初始化 init(int n) n为点数
4. //add\_clause(int x, int xval, int y, int yval);添加条件x = xval or y = yval
5. //solve() 返回是否为2-sat问题
6. //得到：如果mark[i\*2]为true 则事件xi为假 如果mark[i\*2+1]为true 则事件xi为真
7. struct TwoSAT {
8. int n;
9. vector<int> G[maxn\*2];
10. bool mark[maxn\*2];
11. int S[maxn\*2], c;
12. bool dfs(int x) {
13. if (mark[x^1]) return false;//遇到和事实不符的退出
14. if (mark[x]) return true;
15. mark[x] = true;
16. S[c++] = x;
17. for (int i = 0; i < G[x].size(); i++)//dfs下去
18. if (!dfs(G[x][i])) return false;
19. return true;
20. }
21. void init(int n) {
22. this->n = n;
23. for (int i = 0; i < n\*2; i++) G[i].clear();
24. memset(mark, 0, sizeof(mark));
25. }
26. // x = xval or y = yval
27. void add\_clause(int x, int xval, int y, int yval) {
28. x = x \* 2 + xval;
29. y = y \* 2 + yval;
30. G[x^1].push\_back(y);//x^1就是x不满足条件的情况下
31. G[y^1].push\_back(x);
32. }
33. bool solve() {
34. for(int i = 0; i < n\*2; i += 2) {
35. if(!mark[i] && !mark[i+1]) {
36. c = 0;
37. if(!dfs(i)) {//如果dfs是true就进入下一个连通块,否则就删除重做
38. while(c > 0) mark[S[--c]] = false;
39. if(!dfs(i+1)) return false;
40. }
41. }
42. }
43. return true;
44. }
45. } ts;

## Tarjan的求点-双连通分量算法

1. //调用方法：
2. //G为邻接表 下标从0开始
3. //先初始化 init(int n);
4. //add\_edge(u, v); 添加一条u->v的边
5. //调用find\_bcc(); 得到割点和桥
6. //bccno[i]表示i属于哪一个bcc（为0时，为孤立点），割顶的bccno无意义
7. //bcc[i]保存每个分量的点
8. //root为dfs树的根
9. struct Tarjan {
10. struct Edge { int u, v; };
11. int pre[maxn], dfs\_clock, bcc\_cnt, n;
12. int is\_cut[maxn], is\_bridge[maxn][maxn]; //is\_bridge只能使用于简单图
13. int bccno[maxn]; // bccno[i]表示i属于哪一个bcc（为0时，为孤立点），割顶的bccno无意义
14. vector<int> G[maxn], bcc[maxn];//bcc[i]保存每个分量的点
15. stack<Edge> S;
16. void init(int n) {
17. // 调用结束后S保证为空，所以不用清空
18. this->n = n;
19. for (int i = 0; i < n; i++) G[i].clear();
20. memset(is\_cut, 0, sizeof(is\_cut));
21. memset(is\_bridge, 0, sizeof(is\_bridge));
22. memset(pre, 0, sizeof(pre));
23. memset(bccno, 0, sizeof(bccno));
24. dfs\_clock = bcc\_cnt = 0;
25. }
26. void add\_edge(int u, int v) {
27. G[u].push\_back(v);
28. }
29. void find\_bcc() {
30. for (int i = 0; i < n; i++) {//点的下标从0开始
31. if (!pre[i]) dfs(i, -1);
32. }
33. }
34. int dfs(int u, int fa) {
35. int lowu = pre[u] = ++dfs\_clock;
36. int child = 0;
37. for (int i = 0; i < G[u].size(); i++) {
38. int v = G[u][i];
39. Edge e = (Edge){u, v};
40. if (!pre[v]) { // 没有访问过v
41. S.push(e);
42. child++;
43. int lowv = dfs(v, u);
44. lowu = min(lowu, lowv); // 用后代的low函数更新自己
45. if (lowv >= pre[u]) {
46. is\_cut[u] = true;
47. if (lowv > pre[u]) {
48. is\_bridge[u][v] = is\_bridge[v][u] = true;
49. }
50. bcc\_cnt++; bcc[bcc\_cnt].clear();//注意！bcc从1开始编号
51. for (;;) {
52. Edge x = S.top(); S.pop();
53. if (bccno[x.u] != bcc\_cnt) {
54. bcc[bcc\_cnt].push\_back(x.u);
55. bccno[x.u] = bcc\_cnt;
56. }
57. if (bccno[x.v] != bcc\_cnt) {
58. bcc[bcc\_cnt].push\_back(x.v);
59. bccno[x.v] = bcc\_cnt;
60. }
61. if (x.u == u && x.v == v) break;
62. }
63. }
64. continue;
65. }
66. if (pre[v] < pre[u] && v != fa) {
67. S.push(e);
68. lowu = min(lowu, pre[v]); // 用反向边更新自己
69. }
70. }
71. if (fa < 0 && child == 1) is\_cut[u] = 0;
72. return lowu;
73. }
74. } ta;

## Tarjan求强连通分量

1. //执行完 find\_scc后
2. //scc\_cnt为强连通分量树，
3. //sccno[i]为点i所在的scc的编号
4. vector<int> G[maxn];
5. int n, dfn[maxn], low[maxn], sccno[maxn], dfs\_clock, scc\_cnt;
6. stack<int> S;//保存当前SCC中的结点
7. void dfs(int u) {//dfs后可以得到low[u];
8. dfn[u] = low[u] = ++dfs\_clock; S.push(u);
9. for(int i = 0; i < G[u].size(); i++) {
10. int v = G[u][i];
11. if(!dfn[v]) {//如果该点没有被访问过
12. dfs(v);//把它所有的子树都访问一边
13. low[u] = min(low[u], low[v]);
14. continue;
15. }
16. if(!sccno[v]) low[u] = min(low[u], dfn[v]);
17. }
18. if(low[u] == dfn[u]) {//如果成立 则说明u是某scc的第一个点
19. scc\_cnt++;
20. for(;;) {
21. int x = S.top(); S.pop();
22. sccno[x] = scc\_cnt;
23. if(x == u) break;//弹出栈里元素，直到弹出u为止
24. }
25. }
26. }
27. void find\_scc() {
28. dfs\_clock = scc\_cnt = 0;
29. memset(sccno, 0, sizeof(sccno));
30. memset(dfn, 0, sizeof(dfn));
31. for(int i = 0; i < n; i++)
32. if(!dfn[i]) dfs(i);
33. }

## 无向图的割顶和桥判断

1. //调用方法：
2. //G为邻接表 下标从0开始
3. //先初始化 init(int n);
4. //add\_edge(u, v) 添加边u->v
5. //调用get\_cut\_bridge(int root); 得到切点和桥
6. //root为dfs树的根
7. struct CB {
8. int pre[maxn], dfs\_clock;
9. bool is\_cut[maxn], is\_bridge[maxn][maxn]; //只使用于简单图
10. int n, m; //m没有用啊
11. vector<int> G[maxn];
12. void init(int n) {
13. this->n = n;
14. for (int i = 0; i < n; i++) G[i].clear();
15. memset(is\_cut, 0, sizeof(is\_cut));
16. memset(is\_bridge, 0, sizeof(is\_bridge));
17. memset(pre, 0, sizeof(pre));
18. }
19. void get\_cut\_bridge(int root) { dfs(root, -1); }
20. void add\_edge(int u, int v) { G[u].push\_back(v); }
21. int dfs(int u, int fa) {//u在dfs树中的父节点是fa
22. int lowu = pre[u] = ++dfs\_clock;
23. int child = 0;//子节点的数目
24. for (int i = 0; i < G[u].size(); i++) {
25. int v = G[u][i];
26. if (!pre[v]) {//没有访问过v
27. child++;
28. int lowv = dfs(v, u);
29. lowu = min(lowu, lowv); //用后代的low函数更新u的low函数
30. if (lowv >= pre[u]) {//存在后代的low值>= pre[u]则u为割顶
31. is\_cut[u] = true;
32. if (lowv > pre[u]) {
33. is\_bridge[u][v] = is\_bridge[v][u] = true;
34. }
35. }
36. continue;
37. }
38. if (pre[v] < pre[u] && v != fa) {//v == fa不是反向边 而是第二次访问（无向图来回
39. lowu = min(lowu, pre[v]);//用反向边更新u的low函数
40. }
41. }
42. if (fa < 0 && child == 1) is\_cut[u] = 0;//u为根节点 且 孩子唯一时 不是割顶
43. return lowu;
44. }
45. };

## 二分图黑白着色

1. //判断结点u所在的连通分量是否为二分图
2. //并获得黑白二着色（1，2分别为黑白，0为为着色）
3. //调用方法color[u] = 1; bipartite(u);
4. int color[maxn];
5. bool bipartite(int u) {
6. for (int i = 0; i < G[u].size(); i++) {
7. int v = G[u][i];
8. if (color[v] == color[u]) return false;
9. if (!color[v]) {
10. color[v] = 3 - color[u];
11. if (!bipartite(v)) return false;
12. }
13. }
14. return true;
15. }

## 拓扑排序

知道每个数的大小大小关系，给出一个总序列的排序

1. int n, G[maxn][maxn];//点数，G[i][j]表示i<j
2. int c[maxn];
3. vector<int> topo;
4. bool dfs(int u) {
5. c[u] = -1;
6. for (int v = 0; v <= n; v++) if (G[u][v]) {//访问每个点, 如果存在一个比u大的数v
7. if (c[v] < 0) return false;//如果c[v]小于0 说明dfs结束之前访问过（存在环，则没有拓扑序）
8. if (!c[v]) dfs(v);
9. }
10. c[u] = 1;//找到最大的节点，并存入且标记
11. topo.push\_back(u);
12. return true;
13. }
14. bool toposort() {
15. topo.clear();
16. memset(c, 0, sizeof(c));
17. for (int u = 0; u <= n; u++) {
18. if (!c[u]) {
19. if (!dfs(u)) return false;
20. }
21. }
22. reverse(topo.begin(), topo.end());//调用之前topo里面保存从大到小，调用之后从小到大
23. return true;
24. }

## Kruskal求最小生成树

/\*==================================================\*\

| 最小生成森林问题(k 颗树)O(mlogm).

根据Kruskal算法思想，图中的生成树在连完第n-1条边前，都是一个最小生

成森林，每次贪心的选择两个不属于同一连通分量的树（如果连接一个连通分

量，因为不会减少块数，那么就是不合算的）且用最“便宜”的边连起来，连

接n-1次后就形成了一棵MST，n-2次就形成了一个两棵树的最小生成森林，

n-3，……，n-k此后就形成了k颗树的最小生成森林，就是题目要求求解的。

\\*==================================================\*/

1. struct Edge {
2. int u, v, w;//从u到v权为w
3. bool operator < (const Edge& rhs) const {
4. return w < rhs.w;
5. }
6. };
7. struct Kruskal {
8. vector<Edge> e;
9. int n;
10. int pa[maxn];
11. void init(int n) { this->n = n; e.clear(); }
12. void add\_edge(int u, int v, int w) {
13. e.push\_back((Edge){u, v, w});
14. }
15. int find(int x) {
16. return pa[x] != x ? pa[x] = find(pa[x]) : x;
17. }
18. int kruskal() {
19. int ans = 0;
20. for (int i = 0; i < n; i++) pa[i] = i;
21. sort(e.begin(), e.end());
22. for (int i = 0; i < e.size(); i++) {
23. int x = find(e[i].u), y = find(e[i].v);
24. if (x != y) {
25. ans += e[i].w;
26. pa[x] = y;
27. }
28. }
29. return ans;
30. }
31. } kru;

## Dijkstra(pair\_heap\_tag优化)

复杂度：O(E + VlogV)

/调用方法：

//点的下标从0开始

void init(int n); //带入总点数n初始化

void add\_edge(int from, int to, int dist);//from->to 距离为dist

void dijkstra(int s);//得到单源最短路

void get\_shortest\_paths(int s, int\* dist, vector<int>\* paths);

//dist[i]为s到i的距离，paths[i]为s到i的最短路径（经过的结点列表，包括s和t）

// 如果paths[i].size()为0 或 dist[i]为INF是 表示路径不连通

1. #include <ext/pb\_ds/priority\_queue.hpp>
2. using \_\_gnu\_pbds::pairing\_heap\_tag;
3. #define x first
4. #define y second
5. typedef pair<int, int> pii;
6. typedef \_\_gnu\_pbds::priority\_queue<pii, greater<pii>, pairing\_heap\_tag> Heap;
7. typedef Heap::point\_iterator Hit;
8. const int INF = 0x3f3f3f3f;
9. const int maxn = 1000 + 10;
10. const Hit null;
11. struct Edge {
12. int from, to, dist;
13. };
14. struct Dijkstra {
15. int n, m;
16. vector<Edge> edges;
17. vector<int> G[maxn];
18. bool done[maxn]; // 是否已永久标号
19. int d[maxn]; // s到各个点的距离
20. int p[maxn]; // 最短路中的上一条弧
21. Hit iter[maxn]; //i在堆中的迭代器
22. void init(int n) {
23. this->n = n;
24. for(int i = 0; i < n; i++) G[i].clear();
25. edges.clear();
26. }
27. void add\_edge(int from, int to, int dist) {
28. edges.push\_back((Edge){from, to, dist});
29. m = edges.size();
30. G[from].push\_back(m-1);
31. }
32. void dijkstra(int s) {
33. Heap q;
34. for(int i = 0; i < n; i++) {
35. d[i] = INF;
36. iter[i] = null;
37. }
38. d[s] = 0;
39. memset(done, 0, sizeof(done));
40. iter[s] = q.push(pii(0, s));
41. while(!q.empty()) {
42. int u = q.top().y; q.pop();
43. if(done[u]) continue;
44. done[u] = true;
45. for(int i = 0; i < G[u].size(); i++) {
46. Edge& e = edges[G[u][i]];
47. if(d[e.to] > d[u] + e.dist) {
48. d[e.to] = d[u] + e.dist;
49. p[e.to] = G[u][i];
50. if (iter[e.to] != null) {
51. q.modify(iter[e.to], pii(d[e.to], e.to));
52. }
53. else iter[e.to] = q.push(pii(d[e.to], e.to));
54. }
55. }
56. }
57. }
58. // dist[i]为s到i的距离，paths[i]为s到i的最短路径（经过的结点列表，包括s和t）
59. // 如果paths[i].size()为0 或 dist[i]为INF是 表示路径不连通
60. void get\_shortest\_paths(int s, int\* dist, vector<int>\* paths) {
61. dijkstra(s);
62. for(int i = 0; i < n; i++) {
63. dist[i] = d[i];
64. if (d[i] == INF) continue;
65. paths[i].clear();
66. int t = i;
67. paths[i].push\_back(t);
68. while(t != s) {
69. paths[i].push\_back(edges[p[t]].from);
70. t = edges[p[t]].from;
71. }
72. reverse(paths[i].begin(), paths[i].end());
73. }
74. }
75. } dj;

## 普通版本的Dijkstra

1. struct Edge { int from, to, dist; };
2. struct HeapNode {
3. int d, u;
4. bool operator < (const HeapNode& rhs) const {
5. return d > rhs.d;
6. }
7. };
8. struct Dijkstra {
9. int n, m;
10. vector<Edge> edges;
11. vector<int> G[maxn];
12. bool done[maxn]; // 是否已永久标号
13. int d[maxn]; // s到各个点的距离
14. int p[maxn]; // 最短路中的上一条弧
15. void init(int n) {
16. this->n = n;
17. for(int i = 0; i < n; i++) G[i].clear();
18. edges.clear();
19. }
20. void AddEdge(int from, int to, int dist) {
21. edges.push\_back((Edge){from, to, dist});
22. m = edges.size();
23. G[from].push\_back(m-1);
24. }
25. void dijkstra(int s) {
26. priority\_queue<HeapNode> Q;
27. for(int i = 0; i < n; i++) d[i] = INF;
28. d[s] = 0;
29. memset(done, 0, sizeof(done));
30. Q.push((HeapNode){0, s});
31. while(!Q.empty()) {
32. HeapNode x = Q.top(); Q.pop();
33. int u = x.u;
34. if(done[u]) continue;
35. done[u] = true;
36. for(int i = 0; i < G[u].size(); i++) {
37. Edge& e = edges[G[u][i]];
38. if(d[e.to] > d[u] + e.dist) {
39. d[e.to] = d[u] + e.dist;
40. p[e.to] = G[u][i];
41. Q.push((HeapNode){d[e.to], e.to});
42. }
43. }
44. }
45. }
46. // dist[i]为s到i的距离，paths[i]为s到i的最短路径（经过的结点列表，包括s和t）
47. void GetShortestPaths(int s, int\* dist, vector<int>\* paths) {
48. dijkstra(s);
49. for(int i = 0; i < n; i++) {
50. dist[i] = d[i];
51. paths[i].clear();
52. int t = i;
53. paths[i].push\_back(t);
54. while(t != s) {
55. paths[i].push\_back(edges[p[t]].from);
56. t = edges[p[t]].from;
57. }
58. reverse(paths[i].begin(), paths[i].end());
59. }
60. }
61. };

## Bellman Ford

//调用方法：

//点的下标从0开始

void init(int n); //带入总点数n初始化

void add\_edge(int from, int to, double dist);//from->to 距离为dist

bool spfa(int s);//以s为起点 得到单源最短路 如果不存在最短路返回false

bool has\_negative\_cycle();//判断图是否含有负圈，存在返回true

1. struct Edge {
2. int from, to;
3. double dist;
4. };
5. struct BellmanFord {
6. int n, m;
7. vector<Edge> edges;
8. vector<int> G[maxn];
9. bool inq[maxn]; // 是否在队列中
10. double d[maxn]; // s到各个点的距离
11. int p[maxn]; // 最短路中的上一条弧
12. int cnt[maxn]; // 进队次数
13. void init(int n) {
14. this->n = n;
15. for(int i = 0; i < n; i++) G[i].clear();
16. edges.clear();
17. }
18. void add\_edge(int from, int to, double dist) {
19. edges.push\_back((Edge){from, to, dist});
20. m = edges.size();
21. G[from].push\_back(m-1);
22. }
23. bool spfa(int s) {//s为起点 存在最短路返回true
24. queue<int> q;
25. memset(inq, 0, sizeof(inq));
26. memset(cnt, 0, sizeof(cnt));
27. for (int i = 0; i < n; i++) d[i] = INF;
28. d[s] = 0;
29. inq[s] = true;
30. q.push(s);
31. while(!q.empty()) {
32. int u = q.front(); q.pop();
33. inq[u] = false;
34. for(int i = 0; i < G[u].size(); i++) {
35. Edge& e = edges[G[u][i]];
36. if(d[e.to] > d[u] + e.dist) {
37. d[e.to] = d[u] + e.dist;
38. p[e.to] = G[u][i];
39. if(!inq[e.to]) {
40. q.push(e.to);
41. inq[e.to] = true;
42. if(++cnt[e.to] > n) return false;//发现负圈时退出(每个点都帮他更新也不会超过n)
43. }
44. }
45. }
46. }
47. return true;
48. }
49. bool has\_negative\_cycle() {//存在负圈返回true
50. queue<int> q;
51. memset(inq, 0, sizeof(inq));
52. memset(cnt, 0, sizeof(cnt));
53. for(int i = 0; i < n; i++) { //所有点都扔进去，不连通的图也能做
54. d[i] = 0;
55. inq[0] = true;
56. q.push(i);
57. }
58. while(!q.empty()) {
59. int u = q.front(); q.pop();
60. inq[u] = false;
61. for(int i = 0; i < G[u].size(); i++) {
62. Edge& e = edges[G[u][i]];
63. if(d[e.to] > d[u] + e.dist) {
64. d[e.to] = d[u] + e.dist;
65. p[e.to] = G[u][i];
66. if(!inq[e.to]) {
67. q.push(e.to);
68. inq[e.to] = true;
69. if(++cnt[e.to] > n) return true;
70. }
71. }
72. }
73. }
74. return false;
75. }
76. } bf;

## Minimal Steiner Tree

1. /\*
2. Minimal Steiner Tree
3. | G(V, E), A是V的一个子集, 求至少包含A中所有点的最小子树.
4. | 时间复杂度: O(N^3 + N \* 2^A \* (2^A + N))
5. | INIT: d[][]距离矩阵; id[]置为集合A中点的标号;
6. | CALL: steiner(int n, int a);
7. | main()函数解决的题目: Ticket to Ride, POJ 3123
8. | 给4个点对(a1, b1) ... (a4, b4),
9. | 求min(sigma(dist[ai][bi])),其中重复的路段只能算一次.
10. | 这题要找出一个steiner森林, 最后要对森林中树的个数进行枚举
11. 原理：
12. 一个图的最小生成树即这个图上所有点（设集合为G）生成的边权值和最小的树，
13. 我们可以知道这个树上任意两点是可到达的，这个可以用prime或者kruskal实现。
14. 下面，假如我们只要求G的一个真子集里面的所有点连通，
15. 那么我们发现一些边权是没有用的、可以去掉，这样，将指定点集合中的所有点连通，
16. 且总边权值和最小的生成树称谓MinimalSteinerTree（最小斯坦纳树），
17. 可以看出，最小生成树是最小斯坦纳树的一种特殊情况。
18. SteinerTree是组合优化学的著名问题，而且是离散数学中几个NP问题之一，
19. 可是很多离散书中提都不提.....SteinerTree对于求解数学模型，优化最小网络很有用。
20. 求解方法：
21. 首先，我们从prime的思想出发，如果求最小斯坦纳树，那么只要求对应点集为最小生成树即可，
22. 而对应点集的最小生成树和整体最小生成树想比肯定会出现无用点，
23. 所以先用floyd算出两点之间最短距离，之后将所有点都两两连接起来，形成一个完全图，
24. 这样就将整个图缩了一下，之后，枚举所有可能点集，对每个点集求最小生成树，取最小即可。
25. 可是我们会发现，点集的个数是个不可知，而且每次都要做一次prime算法，所以复杂之高难以估算。
26. 于是，我们要改进一下算，转用DP来求解。
27. DP转移方程：
28. dp[mask][i]，其中是以i为根，包含mask点集的最小生成树的权值，mask作为点集可以用二进制进行状态压缩。
29. 在得知dp[mask - 1][1...N]的情况下如何退出dp[mask][1...N]呢?
30. 两个步骤实现：
31. step1：
32. 从点集入手
33. a = min(dp[m1][i] + dp[m2][i])，其中m1 | m2 = mask。
34. m1和m2作为mask的子集，且互补。
35. 在计算子集的时候，可有用for(int sub = mask; sum; sum = (sum - 1) & mask)来枚举子集
36. step2：
37. 从根入手
38. b = min(dp[mask][j] + dp[j][i])
39. 如果mask点集满足且根为j，那么直接填上[j][i]这条边就行了（前面拿floyd已经缩图了。）
40. 程序中，每次都从dp[mask][1...N]中选出最小的一个dp[mask][c]，按这种顺序更新就能保证结果的正确
41. 所以，dp[mask][i] = min(a, b);
42. 代码用法：
43. 首先初始化d数组，不连通的点之间距离为inf memset(d, 0x3f, sizeof(d));
44. d[i][i]=0
45. 如果a<->b距离为c，那么d[a][b]=d[b][a]=c;
46. 要连通的几个点为4,5,9的话，那么在id[0]=4,id[1]=5,id[2]=9。下标从0开始。
47. a表示有几个顶点，是必须选择的。
48. dp[?][i]的第一维，是id数组里的元素的序号的状态压缩。101表示选了id数组的第0个和第3个元素。i则表示根节点。
49. 比如dp[3][4]表示以4为根节点的生成树，包含id[0]与id[2]号元素的图，的最小数值。
50. \\*==================================================\*/
51. #define typec int // type of cost
52. const typec inf = 0x3f3f3f3f; // max of cost
53. int vis[V], id[A]; //id[]: A中点的标号
54. typec d[V][V], dp[1<<A][V];//dp[i][v]: 点v到点集i的最短距离
55. void steiner(int n, int a){
56. int i, j, k, mx, mk, top = (1 << a);
57. //floyd 用完d就坏了
58. for (k = 0; k < n; k++) for (i = 0; i < n; i++)
59. for (j = 0; j < n; j++)
60. if (d[i][j] > d[i][k] + d[k][j])
61. d[i][j] = d[i][k] + d[k][j];
62. for (i = 0; i < a; i++) { // vertex: 0 ~ n-1
63. for (j = 0; j < n; j++)
64. dp[1 << i][j] = d[j][id[i]];
65. }
66. for (i = 1; i < top; i++) {
67. if ( 0 == (i & (i - 1)) ) continue;
68. memset(vis, 0, sizeof(vis));
69. for (k = 0; k < n; k++) { // init
70. for (dp[i][k] = inf, j = 1; j < i; j++)
71. if ((i | j) == i &&
72. dp[i][k] > dp[j][k] + dp[i - j][k])
73. dp[i][k] = dp[j][k] + dp[i - j][k];
74. }
75. for (j = 0; mx = inf, j < n; j++) { // update
76. for (k = 0; k < n; k++)
77. if (dp[i][k] <= mx && 0 == vis[k])
78. mx = dp[i][mk = k];
79. for (k = 0, vis[mk] = 1; k < n; k++)
80. if (dp[i][mk] > dp[i][k] + d[k][mk])
81. dp[i][mk] = dp[i][k] + d[k][mk];
82. }
83. }
84. }
85. int main() {
86. int n, a = 8;
87. steiner(n, a);
88. for (i = 0, b = inf; z = 0, i < 256; b>z ? b=z : b, i++)
89. for (j = 0; y = 0, j < 4; z += !!y \* dp[y][x], j++)
90. for (k = 0; k < 8; k += 2)
91. if ((i >> k & 3) == j)
92. y += 3 << k, x = id[k];
93. return 0;
94. }

## Floyd Warshall(求每两点之间的最短路)

复杂度：O(n3)

1. int d[maxn][maxn];//邻接表 点的下标从0开始
2. for (int i = 0; i < n; i++) {//初始化
3. for (int j = 0; j < n; j++) {
4. d[i][j] = (i == j ? 0 : INF);
5. }
6. }
7. for (int k = 0; k < n; k++) {
8. for (int i = 0; i < n; i++) {
9. for (int j = 0; j < n; j++) {
10. d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
11. }
12. }
13. }

## Floyd 求最小环

1. /\*==================================================\*\
2. | Floyd 求最小环
3. 在floyd的同时，顺便算出最小环
4. 一个环中的最大结点为k(编号最大), 与他相连的两个点为i, j, 这个环的最
5. 短长度为g[i][k]+g[k][j]+i到j的路径中所有结点编号都小于k的最短路
6. 径长度. 根据floyd的原理, 在最外层循环做了k-1次之后, dist[i][j]则
7. 代表了i到j的路径中所有结点编号都小于k的最短路径
8. 综上所述,该算法一定能找到图中最小环.
9. 调用方法：
10. 输入n m（点数边数）
11. 在输入m组 x y l 分别为 点x到点y距离为l
12. （输入下标从1开始， 处理下标从0开始）
13. \\*==================================================\*/
14. const int N = 110;
15. int n, m; // n:节点个数, m:边的个数
16. int g[N][N]; // 无向图 存边长
17. int dist[N][N]; // 最短路径
18. int r[N][N]; // r[i][j]: i到j的最短路径的第一步
19. int out[N], ct; // 记录最小环
20. int solve(int i, int j, int k){// 记录最小环
21. ct = 0;
22. while ( j != i ){
23. out[ct++] = j;
24. j = r[i][j];
25. }
26. out[ct++] = i; out[ct++] = k;
27. return 0;
28. }
29. int main() {
30. while(scanf("%d%d", &n, &m) != EOF){
31. int i, j, k;
32. for (i=0; i < n; i++) for (j=0; j < n; j++) {
33. g[i][j] = INF; r[i][j] = i; }
34. for (i=0; i < m; i++) {
35. int x, y, l;
36. scanf("%d%d%d", &x, &y, &l); --x; --y;
37. if (l < g[x][y]) g[x][y] = g[y][x] = l;
38. }
39. memmove(dist, g, sizeof(dist));
40. int Min = INF; // 最小环
41. for (k=0; k < n; k++) {//Floyd
42. // 一个环中的最大结点为k(编号最大)
43. for (i=0; i < k; i++) if (g[k][i] < INF) for (j=i+1; j < k; j++)
44. if (dist[i][j]<INF && g[k][j]<INF && Min>dist[i][j]+g[k][i]+g[k][j]) {
45. Min = dist[i][j]+g[k][i]+g[k][j];
46. solve(i, j, k); // 记录最小环
47. }
48. for (i=0; i < n; i++) if (dist[i][k] < INF) for (j=0; j < n; j++)
49. if (dist[k][j] < INF && dist[i][j] > dist[i][k]+dist[k][j]) {
50. dist[i][j] = dist[i][k]+dist[k][j];
51. r[i][j] = r[k][j];
52. }
53. }
54. if (Min < INF) {
55. for (ct--; ct >= 0; ct--) {
56. printf("%d", out[ct]+1);
57. if (ct) printf(" ");
58. }
59. }
60. else printf("No solution.");
61. printf("\n");
62. }
63. return 0;
64. }

# ------------dp相关：

## 最大子串匹配

1. //最大子串匹配,复杂度 O(mn)
2. //(两个字符串中找相同的子序列)
3. //返回最大匹配值,传入两个串和串的长度,重载返回一个最大匹配
4. //注意做字符串匹配是串末的'\0'没有置!
5. //可更改元素类型,更换匹配函数和匹配价值函数
6. #define MAXN 100
7. #define max(a,b) ((a)>(b)?(a):(b))
8. #define \_match(a,b) ((a)==(b))
9. #define \_value(a,b) 1
10. typedef int elem\_t;
11. int str\_match(int m,elem\_t\* a,int n,elem\_t\* b){
12. int match[MAXN+1][MAXN+1],i,j;
13. memset(match,0,sizeof(match));
14. for (i=0;i<m;i++)
15. for (j=0;j<n;j++)
16. match[i+1][j+1]=max(max(match[i][j+1],match[i+1][j]),
17. (match[i][j]+\_value(a[i],b[i]))\*\_match(a[i],b[j]));
18. return match[m][n];
19. }
20. int str\_match(int m,elem\_t\* a,int n,elem\_t\* b,elem\_t\* ret){
21. int match[MAXN+1][MAXN+1],last[MAXN+1][MAXN+1],i,j,t;
22. memset(match,0,sizeof(match));
23. for (i=0;i<m;i++)
24. for (j=0;j<n;j++){
25. match[i+1][j+1]=(match[i][j+1]>match[i+1][j]?match[i][j+1]:match[i+1][j]);
26. last[i+1][j+1]=(match[i][j+1]>match[i+1][j]?3:1);
27. if ((t=(match[i][j]+\_value(a[i],b[i]))\*\_match(a[i],b[j]))>match[i+1][j+1])
28. match[i+1][j+1]=t,last[i+1][j+1]=2;
29. }
30. for (;match[i][j];i-=(last[t=i][j]>1),j-=(last[t][j]<3))
31. ret[match[i][j]-1]=(last[i][j]<3?a[i-1]:b[j-1]);
32. return match[m][n];
33. }

## 最大子段和

1. //求最大子段和,复杂度 O(n)
2. //传入串长 n 和内容 list[]
3. //返回最大子段和,重载返回子段位置(maxsum=list[start]+...+list[end])
4. //可更改元素类型
5. typedef int elem\_t;
6. elem\_t maxsum(int n,elem\_t\* list){
7. elem\_t ret,sum=0;
8. int i;
9. for (ret=list[i=0];i<n;i++)
10. sum=(sum>0?sum:0)+list[i],ret=(sum>ret?sum:ret);
11. return ret;
12. }
13. elem\_t maxsum(int n,elem\_t\* list,int& start,int& end){
14. elem\_t ret,sum=0;
15. int s,i;
16. for (ret=list[start=end=s=i=0];i<n;i++,s=(sum>0?s:i))
17. if ((sum=(sum>0?sum:0)+list[i])>ret)
18. ret=sum,start=s,end=i;
19. return ret;
20. }

## 最大子阵和

1. //求最大子阵和,复杂度 O(n^3)
2. //传入阵的大小 m,n 和内容 mat[][]
3. //返回最大子阵和,重载返回子阵位置(maxsum=list[s1][s2]+...+list[e1][e2])
4. //可更改元素类型
5. #define MAXN 100
6. typedef int elem\_t;
7. elem\_t maxsum(int m,int n,elem\_t mat[][MAXN]){
8. elem\_t matsum[MAXN][MAXN+1],ret,sum;
9. int i,j,k;
10. for (i=0;i<m;i++)
11. for (matsum[i][j=0]=0;j<n;j++)
12. matsum[i][j+1]=matsum[i][j]+mat[i][j];
13. for (ret=mat[0][j=0];j<n;j++)
14. for (k=j;k<n;k++)
15. for (sum=0,i=0;i<m;i++)
16. sum=(sum>0?sum:0)+matsum[i][k+1]-matsum[i][j],ret=(sum>ret?sum:ret);
17. return ret;
18. }
19. elem\_t maxsum(int m,int n,elem\_t mat[][MAXN],int& s1,int& s2,int& e1,int& e2){
20. elem\_t matsum[MAXN][MAXN+1],ret,sum;
21. int i,j,k,s;
22. for (i=0;i<m;i++)
23. for (matsum[i][j=0]=0;j<n;j++)
24. matsum[i][j+1]=matsum[i][j]+mat[i][j];
25. for (ret=mat[s1=e1=0][s2=e2=j=0];j<n;j++)
26. for (k=j;k<n;k++)
27. for (sum=0,s=i=0;i<m;i++,s=(sum>0?s:i))
28. if ((sum=(sum>0?sum:0)+matsum[i][k+1]-matsum[i][j])>ret)
29. ret=sum,s1=s,s2=i,e1=j,e2=k;
30. return ret;
31. }

## 最长单调子序列

1. //最长单调子序列,复杂度 O(nlogn)
2. //注意最小序列覆盖和最长序列的对应关系,例如
3. //"define \_cp(a,b) ((a)>(b))"求解最长严格递减序列,则
4. //"define \_cp(a,b) (!((a)>(b)))"求解最小严格递减序列覆盖
5. //可更改元素类型和比较函数
6. /\*
7. Dilworth定理:
8. 偏序集：设R为非空集合A上的关系，如果R是自反的、反对称的和可传递的，则称R为A上的偏序关系，简称偏序，通常记作≤（小于等于号）
9. 偏序集的定义是
10. 偏序是在集合X上的二元关系≤(这只是个抽象符号，不是“小于或等于”)，它满足自反性、反对称性和传递性。
11. 即，对于X中的任意元素a,b和c，有:
12. 自反性：a≤a;
13. 反对称性：如果a≤b且b≤a，则有a=b;
14. 传递性：如果a≤b且b≤c，则a≤c 。
15. 带有偏序关系的集合称为偏序集。
16. 令(X,≤)是一个偏序集，对于集合中的两个元素a、b，如果有a≤b或者b≤a，则称a和b是可比的，否则a和b不可比。
17. 例：(A,≤)是偏序集，其中A={1,2,3,4,5}，其中≤是整除关系，那么对任意的x∈p都有1≤x,所以1和1,2,3,4,5都是可比的，
18. 但是2不能整除3，且3不能整除2，所以2和3是不可比的
19. 链（chain）是一个偏序集S的全序子集（所谓全序是指任意两个元素可比较）
20. 反链（antichain）是一个偏序集S的子集，其中任意两个元素不可比较（不满足<=，这是偏序关系）
21. 极大（maximal）链:对一个链C，如果找不到另一个链C'，使得C是C'的真子集，那么称链C是极大的
22. 极大（maximal）反链：对一个反链A，如果找不到另一个反链A'，使得A是A'的真子集，那么称反链A是极大的
23. 偏序集S的最大链的大小称为偏序集S的高度（height）
24. 偏序集S的最大反链的大小称为偏序集S的宽度（width）
25. 定理1 令（X,≤）是一个有限偏序集，并令r是其最大链的大小。则X可以被划分成r个但不能再少的反链。
26. 其对偶定理称为Dilworth定理：
27. 定理2 令（X,≤）是一个有限偏序集，并令m是反链的最大的大小。则X可以被划分成m个但不能再少的链。
28. \*/
29. #define MAXN 10000
30. #define \_cp(a,b) ((a)>(b))
31. typedef int elem\_t;
32. int subseq(int n,elem\_t\* a){
33. int b[MAXN],i,l,r,m,ret=0;
34. for (i=0;i<n;b[l]=i++,ret+=(l>ret))
35. for (m=((l=1)+(r=ret))>>1;l<=r;m=(l+r)>>1)
36. if (\_cp(a[b[m]],a[i])) l=m+1;
37. else r=m-1;
38. return ret;
39. }
40. int subseq(int n,elem\_t\* a,elem\_t\* ans){
41. int b[MAXN],p[MAXN],i,l,r,m,ret=0;
42. for (i=0;i<n;p[b[l]=i++]=b[l-1],ret+=(l>ret))
43. for (m=((l=1)+(r=ret))>>1;l<=r;m=(l+r)>>1)
44. if (\_cp(a[b[m]],a[i])) l=m+1;
45. else r=m-1;
46. for (m=b[i=ret];i;ans[--i]=a[m],m=p[m]);
47. return ret;
48. }

## 序列逆序对数

1. //序列逆序对数,复杂度 O(nlogn)
2. //传入序列长度和内容,返回逆序对数
3. //可更改元素类型和比较函数
4. #define MAXN 1000000
5. int \_tmp[MAXN];
6. int inv(int n, int\* a) {
7. int l=n>>1,r=n-l,i,j;
8. int ret=(r>1?(inv(l,a)+inv(r,a+l)):0);
9. for (i=j=0;i<=l;\_tmp[i+j]=a[i],i++)
10. for (ret+=j;j<r&&(i==l||!(a[i] <= a[l+j]));\_tmp[i+j]=a[l+j],j++);
11. memcpy(a,\_tmp,sizeof(int)\*n);
12. return ret;
13. }

## 最长公共递增子序列

1. // 最长公共递增子序列， 时间复杂度 O(n^2 \* logn)，空间 O(n^2)
2. /\*\*
3. \* n 为 a 的大小, m 为 b 的大小
4. \* 结果在 ans 中
5. \* 求解最长严格递增序列
6. 返回值为ans的长度
7. \*/
8. #define MAXN 1000
9. #define \_cp(a,b) ((a)<(b))
10. typedef int elem\_t;
11. elem\_t DP[MAXN][MAXN];
12. int num[MAXN], p[1<<20];
13. int LIS(int n, elem\_t \*a, int m, elem\_t \*b, elem\_t \*ans){
14. int i, j, l, r, k;
15. DP[0][0] = 0;
16. num[0] = (b[0] == a[0]);
17. for(i = 1; i < m; i++) {
18. num[i] = (b[i] == a[0]) || num[i-1];
19. DP[i][0] = 0;
20. }
21. for(i = 1; i < n; i++){
22. if(b[0] == a[i] && !num[0]) {
23. num[0] = 1;
24. DP[0][0] = i<<10;
25. }
26. for(j = 1; j < m; j++){
27. for(k=((l=0)+(r=num[j-1]-1))>>1; l<=r; k=(l+r)>>1)
28. if(\_cp(a[DP[j-1][k]>>10], a[i])) l=k+1;
29. else r=k-1;
30. if(l < num[j-1] && i == (DP[j-1][l]>>10) ){
31. if(l >= num[j]) DP[j][num[j]++] = DP[j-1][l];
32. else DP[j][l] = \_cp(a[DP[j][l]>>10],a[i]) ? DP[j][l] : DP[j-1][l];
33. }
34. if(b[j] == a[i]){
35. for(k=((l=0)+(r=num[j]-1))>>1; l<=r; k=(l+r)>>1)
36. if(\_cp(a[DP[j][k]>>10], a[i])) l=k+1;
37. else r=k-1;
38. DP[j][l] = (i<<10) + j;
39. num[j] += (l>=num[j]);
40. p[DP[j][l]] = l ? DP[j][l-1] : -1;
41. }
42. }
43. }
44. for (k=DP[m-1][i=num[m-1]-1];i>=0;ans[i--]=a[k>>10],k=p[k]);
45. return num[m-1];
46. }

# ------数学：

## 清华定积分

1. /\* Romberg 求定积分
2. 输入：积分区间[a,b]，被积函数 f(x,y,z)
3. 输出：积分结果
4. f(x,y,z)示例：
5. double f0( double x, double l, double t )
6. {
7. return sqrt(1.0+l\*l\*t\*t\*cos(t\*x)\*cos(t\*x));
8. } \*/
9. double f0(double x, double l, double t)
10. {
11. return sqrt(1.0+l\*l\*t\*t\*cos(t\*x)\*cos(t\*x));
12. }
13. double Integral(double a, double b, double (\*f)(double x, double y, double z), double eps, double l, double t);
14. double Romberg (double a, double b, double (\*f)(double x, double y, double z), double eps, double l, double t)
15. {
16. #define MAX\_N 1000
17. int i, j, temp2, min;
18. double h, R[2][MAX\_N], temp4;
19. for (i=0; i<MAX\_N; i++) {
20. R[0][i] = 0.0;
21. R[1][i] = 0.0;
22. }
23. h = b-a;
24. min = (int)(log(h\*10.0)/log(2.0)); //h should be at most 0.1
25. R[0][0] = ((\*f)(a, l, t)+(\*f)(b, l, t))\*h\*0.50;
26. i = 1; temp2 = 1;
27. while (i<MAX\_N) {
28. i++;
29. R[1][0] = 0.0;
30. for (j=1; j<=temp2; j++) R[1][0] += (\*f)(a+h\*((double)j-0.50), l, t);
31. R[1][0] = (R[0][0] + h\*R[1][0])\*0.50;
32. temp4 = 4.0;
33. for (j=1; j<i; j++) {
34. R[1][j] = R[1][j-1] + (R[1][j-1]-R[0][j-1])/(temp4-1.0);
35. temp4 \*= 4.0;
36. }
37. if ((fabs(R[1][i-1]-R[0][i-2])<eps)&&(i>min)) return R[1][i-1];
38. h \*= 0.50; temp2 \*= 2;
39. for (j=0; j<i; j++) R[0][j] = R[1][j];
40. }
41. return R[1][MAX\_N-1];
42. }
43. double Integral(double a, double b, double (\*f)(double x, double y, double z), double eps, double l, double t)
44. {
45. #define pi 3.1415926535897932
46. int n;
47. double R, p, res;
48. n = (int)(floor)(b \* t \* 0.50 / pi);
49. p = 2.0 \* pi / t;
50. res = b - (double)n \* p;
51. if (n) R = Romberg (a, p, f0, eps/(double)n, l, t);
52. R = R \* (double)n + Romberg( 0.0, res, f0, eps, l, t );
53. return R/100.0;
54. }

## 上交定积分

1. //f为函数 a,b为区间
2. template<class T>
3. double romberg(const T&f, double a, double b, double eps=1e-8){
4. vector<double>t; double h=b-a,last,curr; int k=1,i=1;
5. t.push\_back(h\*(f(a)+f(b))/2); // 梯形
6. do{ last=t.back(); curr=0; double x=a+h/2;
7. for(int j=0;j<k;++j) curr+=f(x),x+=h;
8. curr=(t[0]+h\*curr)/2; double k1=4.0/3.0,k2=1.0/3.0;
9. for(int j=0;j<i;j++){ double temp=k1\*curr-k2\*t[j];
10. t[j]=curr; curr=temp; k2/=4\*k1-k2; k1=k2+1; // 防止溢出
11. } t.push\_back(curr); k\*=2; h/=2; i++;
12. } while(std::fabs(last-curr)>eps);
13. return t.back();
14. }

## 牛顿法解多项式的根

1. /\* 牛顿法解多项式的根
2. 输入：多项式系数 c[]，多项式度数 n，求在[a,b]间的根
3. c[i]为x^i前面的系数
4. 输出：一个根
5. 要求保证[a,b]间有根
6. \*/
7. double f(int m, double c[], double x) {
8. double p = c[m];
9. for (int i=m; i>0; i--) p = p\*x + c[i-1];
10. return p;
11. }
12. int newton(double x0, double \*r, double c[], double cp[],
13. int n, double a, double b, double eps) {
14. int MAX\_ITERATION = 1000;
15. int i = 1;
16. double x1, x2, fp, eps2 = eps/10.0;
17. x1 = x0;
18. while (i < MAX\_ITERATION) {
19. x2 = f(n, c, x1); fp = f(n-1, cp, x1);
20. if ((fabs(fp)<1e-9) && (fabs(x2)>1.0)) return 0;
21. x2 = x1 - x2/fp;
22. if (fabs(x1-x2)<eps2) {
23. if (x2<a || x2>b) return 0;
24. \*r = x2;
25. return 1;
26. }
27. x1 = x2; i++;
28. }
29. return 0;
30. }
31. double Polynomial\_Root(double c[], int n, double a, double b, double eps) {
32. double \*cp, root;
33. cp = (double \*)calloc(n, sizeof(double));
34. for (int i=n-1; i>=0; i--) cp[i] = (i+1)\*c[i+1];
35. if (a>b) { root = a; a = b; b = root; }
36. if ((!newton(a, &root, c, cp, n, a, b, eps))
37. && (!newton(b, &root, c, cp, n, a, b, eps)))
38. newton((a+b)\*0.5, &root, c, cp, n, a, b, eps);
39. free(cp);
40. if (fabs(root)<eps) return fabs(root);
41. return root;
42. }

## 矩阵乘法

1. const int mat\_size = maxn;
2. struct Matrix {
3. LL a[mat\_size][mat\_size];
4. int x, y;//长宽
5. Matrix() { memset(a,0,sizeof(a)); } //返回0矩阵
6. Matrix(int x, int y) : x(x), y(y) { memset(a,0,sizeof(a)); }//返回0矩阵，并且x,y赋值
7. Matrix(int n) : x(n), y(n) { //返回n\*n的单位矩阵
8. memset(a,0,sizeof(a));
9. for (int i = 0; i < n; i++) a[i][i]=1;
10. }
11. LL\* operator [] (int c) { return a[c]; }
12. Matrix operator \* (const Matrix& rhs) {//矩阵乘法
13. Matrix ret;
14. for (int i = 0; i < x; i++)
15. for (int j = 0; j < rhs.y; j++)
16. for (int k = 0; k < y; k++)
17. ret.a[i][j] = (ret.a[i][j] + a[i][k] \* rhs.a[k][j] % MOD) % MOD;
18. ret.x = x;
19. ret.y = rhs.y;
20. return ret;
21. }
22. Matrix operator ^ (LL b) {//矩阵A的b次方
23. Matrix A(\*this), ret(x);
24. while (b) {
25. if(b & 1) ret = ret \* A;
26. b >>= 1; A = A \* A;
27. }
28. return ret;
29. }
30. Matrix operator & (LL b) {//A^0 + A^1+A^2+A^3+++A^n，其中A是矩阵。最后返回的就是一个矩阵
31. Matrix ret(\*this);
32. for (int i = ret.x; i < ret.x \* 2; i++) {
33. ret.a[i-ret.x][i]= 1;
34. ret.a[i][i] = 1;
35. }
36. ret.x <<= 1; ret.y <<= 1;
37. ret = ret^b;
38. ret.x >>= 1; ret.y >>= 1;
39. for (int i = 0; i < ret.x; i++)
40. for (int j = 0; j < ret.y; j++)
41. ret.a[i][j] += ret.a[i][j + ret.x];
42. return ret;
43. }
44. void pg() {
45. for (int i = 0; i < x; i++) {
46. for (int j = 0; j < y; j++)
47. cout<<a[i][j]<<" ";
48. cout<<endl;
49. }
50. cout<<endl;
51. }
52. } m;

## 求n以内的所有素数

1. const int maxn = 10000000 + 10;
2. const int maxp = 700000;//maxn/math.log(maxn)
3. bool prime[maxn];
4. int p[maxp];
5. int get\_prime(int n)
6. {
7. int k = 0;
8. prime[1] = 1;
9. for(LL i = 2; i < n; i++)
10. {
11. if(!prime[i])
12. {
13. p[k++] = i;
14. for(LL j = i \* i; j < n; j += i)
15. prime[j] = true;
16. }
17. }
18. return k;
19. }

## 唯一分解(e[i]为p[i]的个数)

1. int e[maxp];
2. int len = get\_prime(1000);
3. for (int i = 0; i < len; ++i) {
4. while (n % p[i] == 0) { n /= p[i]; ++e[i]; }
5. if (n == 1) break;
6. }

## 欧几里德

1. LL gcd(LL a, LL b) {
2. return b == 0 ? a : gcd(b, a % b);
3. }

## 扩展欧几里德 ax+by=d, d = gcd(a, b)

1. void gcd(LL a, LL b, LL &d, LL &x, LL &y) {
2. if (!b) { d = a; x = 1; y = 0; }
3. else { gcd(b, a % b, d, y, x); y -= x \* (a / b); }
4. }

## 计算模n下a的逆，不存在返回1

1. LL inv(LL a, LL n) {
2. LL d, x, y;
3. gcd(a, n, d, x, y);//扩展欧几里德
4. return d == 1 ? (x + n) % n : 1;
5. }

## 大整数取模

1. scanf("%s%d", n, &m);
2. int len = strlen(n);
3. LL ans = 0;
4. for (int i = 0; i < len; ++i)
5. ans = (ans \* 10 + n[i] - '0') % m;
6. cout << ans;

## 幂取模，返回a^p mod n(0 <= a < n)

1. LL pow\_mod(LL a, LL b, LL p)//a^b % p
2. {
3. LL r = 1;
4. a %= p;
5. while (b)
6. {
7. if (b & 1) r = r \* a % p;
8. b >>= 1;
9. a = a \* a % p;
10. }
11. return r;
12. }

## 中国剩余定理

1. //n个方程: x = a[i](mod m[i]) (0 <= i < n)
2. LL china(int n, int \*a, int \*m) {
3. LL M = 1, d, y, x = 0;
4. for (int i = 0; i < n; ++i) M \*= m[i];
5. for (int i = 0; i < n; ++i ) {
6. LL w = M / m[i];
7. gcd(m[i], w, d, d, y);
8. x = (x + y \* w \* a[i]) % M;
9. }
10. return (x + M) % M;
11. }

## 求解模方程 a^x = b(mod n)。n为素数，无解返回-1

1. int logMod(int a, int b, int n) {
2. int m, v, e = 1;
3. m = (int)sqrt(n + 0.5);
4. v = inv(powMod(a, m, n), n);
5. map<int, int> x;
6. x[1] = 0;
7. for (int i = 1; i < m; ++i) {
8. e = mulMod(e, a, n);
9. if (!x.count(e)) x[e] = i;
10. }
11. for (int i = 0; i < m; ++i) {
12. if (x.count(b)) return i \* m + x[b];
13. b = mulMod(b, v, n);
14. }
15. return -1;
16. }

## 求二项式定理中各项的系数

1. C[0] = 1;
2. for (int i = 1; i <= n; ++i)  C[i] = C[i - 1] \* (n - i + 1) / i;

## 求出所有i,j小于n的c(i, j)

1. for(int i = 0; i <= n; i++) {
2. C[i][0] = C[i][i] = 1;
3. for(int j = 1; j < i; j++)
4. C[i][j] = (C[i-1][j] + C[i-1][j-1]);
5. }

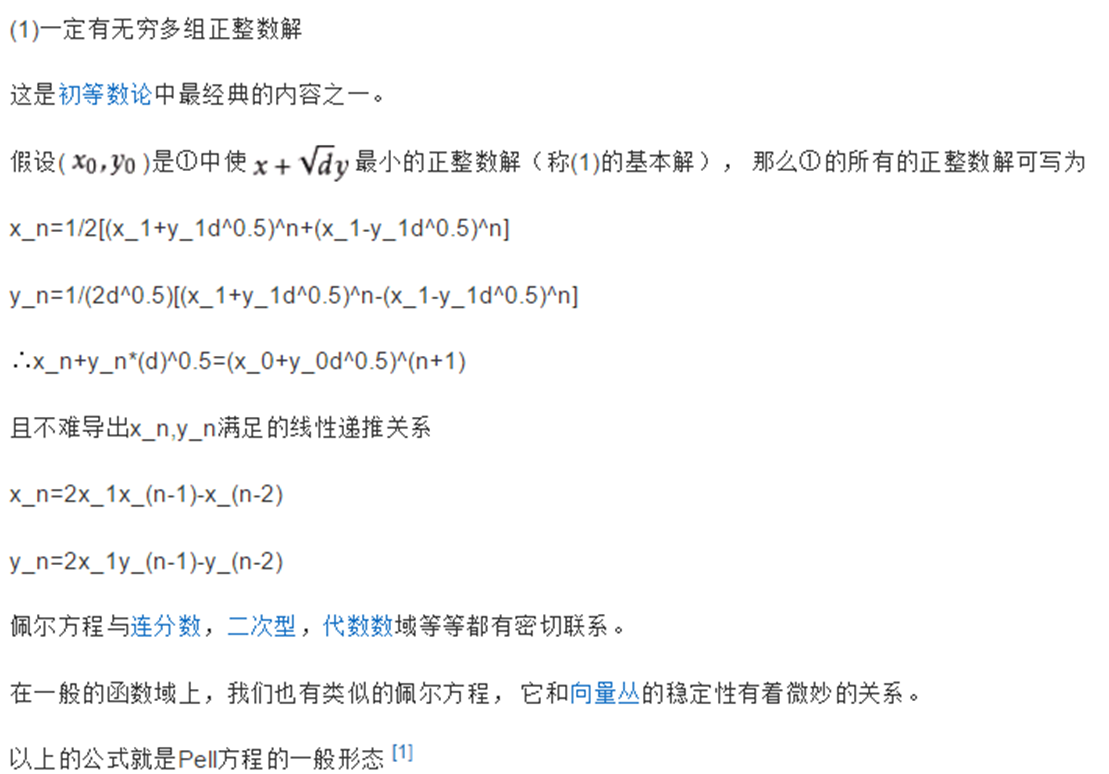
## n的欧拉函数值(不超过n且与n互素的正整数个数)

1. int eulerPhi(int n) {
2. int m = (int)sqrt(n + 0.5), ans = n;
3. for (int i = 2; i <= m; ++i) if (n % i == 0) {
4. ans = ans / i \* (i - 1);
5. while (n % i == 0) n /= i;
6. }
7. if (n > 1) ans = ans / n \* (n - 1);
8. return ans;
9. }

## 1~n中所有数的欧拉函数值

1. int phi[maxn];
2. void get\_phi\_table(int n) {
3. for (int i = 1; i < n; i++) phi[i] = i;
4. for (int i = 2; i <= n; ++i) if (phi[i] == i)
5. for (int j = i; j <= n; j += i)
6. phi[j] = phi[j] - phi[j] / i;
7. }

## Pell方程



求 x^2-ny^2=1 的最小正整数根

1. // 求 x^2-ny^2=1 的最小正整数根, n 不是完全平方数
2. #define sqr(x) ((x)\*(x))
3. ULL A,B,p[maxn],q[maxn],a[maxn],g[maxn],h[maxn];
4. int main() {
5. for (int test=1, n;scanf("%d",&n) && n;++test) {
6. printf("Case %d: ",test);
7. if (fabs(sqrt(n)-floor(sqrt(n)+1e-7))<=1e-7) {
8. int a=(int)(floor(sqrt(n)+1e-7)); printf("%d %d\n",a,1);
9. } else {
10. p[1]=q[0]=h[1]=1;p[0]=q[1]=g[1]=0;
11. a[2]=(int)(floor(sqrt(n)+1e-7));
12. for (int i=2;i;++i) {
13. g[i]=-g[i-1]+a[i]\*h[i-1]; h[i]=(n-sqr(g[i]))/h[i-1];
14. a[i+1]=(g[i]+a[2])/h[i]; p[i]=a[i]\*p[i-1]+p[i-2];
15. q[i]=a[i]\*q[i-1]+q[i-2];
16. if (sqr((ULL)(p[i]))-n\*sqr((ULL)(q[i]))==1){
17. A=p[i];B=q[i];break;
18. }
19. }
20. cout << A << ' ' << B <<endl;
21. }
22. }

## 求x^2 mod p = a 的解

1. //接口是solve
2. LL pow\_mod(LL a, LL b, LL p)//a^b % p
3. {
4. LL r = 1; a %= p;
5. while (b) {
6. if (b & 1) r = r \* a % p;
7. b >>= 1; a = a \* a % p;
8. }
9. return r;
10. }
11. void calcH(int &t, int &h, const int p) {
12. int tmp = p - 1; for (t = 0; (tmp & 1) == 0; tmp /= 2) t++; h = tmp;
13. }
14. // solve equation x^2 mod p = a
15. bool solve(int a, int p, int &x, int &y) {
16. srand(19920225);
17. if (p == 2) { x = y = 1; return true; }
18. int p2 = p / 2, tmp = pow\_mod(a, p2, p);
19. if (tmp == p - 1) return false;
20. if ((p + 1) % 4 == 0) {
21. x = pow\_mod(a, (p + 1) / 4, p); y = p - x; return true;
22. } else {
23. int t, h, b, pb; calcH(t, h, p);
24. if (t >= 2) {
25. do {b = rand() % (p - 2) + 2;
26. } while (pow\_mod(b, p / 2, p) != p - 1);
27. pb = pow\_mod(b, h, p);
28. } int s = pow\_mod(a, h / 2, p);
29. for (int step = 2; step <= t; step++) {
30. int ss = (((long long)(s \* s) % p) \* a) % p;
31. for (int i = 0; i < t - step; i++) ss = ((long long)ss \* ss) % p;
32. if (ss + 1 == p) s = (s \* pb) % p; pb = ((long long)pb \* pb) % p;
33. } x = ((long long)s \* a) % p; y = p - x;
34. } return true;
35. }

## 线性规划（单纯形法）

注意转化格式：

最大化 

满足约数条件



1. //用法细节见主函数
2. #define ULL unsigned long long
3. #define EPS 1e-8
4. //求max{cx | Ax ≤ b, x ≥ 0}的解
5. typedef vector<double> VD;
6. VD simplex(vector<VD> A, VD b, VD c) {
7. int n = A.size(), m = A[0].size() + 1, r = n, s = m - 1;
8. vector<VD> D(n + 2, VD(m + 1, 0)); vector<int> ix(n + m);
9. for (int i = 0; i < n + m; ++ i) ix[i] = i;
10. for (int i = 0; i < n; ++ i) {
11. for (int j = 0; j < m - 1; ++ j) D[i][j] = -A[i][j];
12. D[i][m - 1] = 1; D[i][m] = b[i];
13. if (D[r][m] > D[i][m]) r = i;
14. }
15. for (int j = 0; j < m - 1; ++ j) D[n][j] = c[j];
16. D[n + 1][m - 1] = -1;
17. for (double d; ; ) {
18. if (r < n) {
19. int t = ix[s]; ix[s] = ix[r + m]; ix[r + m] = t;
20. D[r][s] = 1.0 / D[r][s]; vector<int> speedUp;
21. for (int j = 0; j <= m; ++ j) if (j != s) {
22. D[r][j] \*= -D[r][s];
23. if(D[r][j]) speedUp.push\_back(j);
24. }
25. for (int i = 0; i <= n + 1; ++ i) if (i != r) {
26. for(int j = 0; j < speedUp.size(); ++ j)
27. D[i][speedUp[j]] += D[r][speedUp[j]] \* D[i][s];
28. D[i][s] \*= D[r][s];
29. }} r = -1; s = -1;
30. for (int j = 0; j < m; ++ j) if (s < 0 || ix[s] > ix[j])
31. if (D[n + 1][j] > EPS || (D[n + 1][j] > -EPS && D[n][j] > EPS)) s = j;
32. if (s < 0) break;
33. for (int i = 0; i < n; ++ i) if (D[i][s] < -EPS)
34. if (r < 0 || (d = D[r][m] / D[r][s] - D[i][m] / D[i][s]) < -EPS
35. || (d < EPS && ix[r + m] > ix[i + m])) r = i;
36. if (r < 0) return VD(); // 无边界
37. }
38. if (D[n + 1][m] < -EPS) return VD(); // 无解
39. VD x(m - 1);
40. for (int i = m; i < n + m; ++ i) if (ix[i] < m - 1) x[ix[i]] = D[i - m][m];
41. return x; // 最优值在 D[n][m]
42. }
43. int main() {
44. VD a({3,4,1}), b({1,3,2}), d({2,1}),e({3,6,2});//qiu max
45. vector<VD>haha({a,b});
46. VD S = simplex(haha, d, e);
47. for (auto x : S) cout<<x<<endl;
48. }

# ---------数据结构：

## 带花树

1. //带花树：求普通图匹配
2. //下标从0开始，link为图，n为n个点
3. vector<int> link[maxn];
4. int n,match[maxn],Queue[maxn],head,tail;
5. int pred[maxn],base[maxn],start,finish,newbase;
6. bool InQueue[maxn],InBlossom[maxn];
7. void push(int u){ Queue[tail++]=u;InQueue[u]=true; }
8. int pop(){ return Queue[head++]; }
9. int FindCommonAncestor(int u,int v){
10. bool InPath[maxn];
11. for(int i=0;i<n;i++) InPath[i]=0;
12. while(true){ u=base[u];InPath[u]=true;if(u==start) break;u=pred[match[u]]; }
13. while(true){ v=base[v];if(InPath[v]) break;v=pred[match[v]]; }
14. return v;
15. }
16. void ResetTrace(int u){
17. int v;
18. while(base[u]!=newbase){
19. v=match[u];
20. InBlossom[base[u]]=InBlossom[base[v]]=true;
21. u=pred[v];
22. if(base[u]!=newbase) pred[u]=v;
23. }
24. }
25. void BlossomContract(int u,int v){
26. newbase=FindCommonAncestor(u,v);
27. for (int i=0;i<n;i++)
28. InBlossom[i]=0;
29. ResetTrace(u);ResetTrace(v);
30. if(base[u]!=newbase) pred[u]=v;
31. if(base[v]!=newbase) pred[v]=u;
32. for(int i=0;i<n;++i)
33. if(InBlossom[base[i]]){
34. base[i]=newbase;
35. if(!InQueue[i]) push(i);
36. }
37. }
38. bool FindAugmentingPath(int u){
39. bool found=false;
40. for(int i=0;i<n;++i) pred[i]=-1,base[i]=i;
41. for (int i=0;i<n;i++) InQueue[i]=0;
42. start=u;finish=-1; head=tail=0; push(start);
43. while(head<tail){
44. int u=pop();
45. for(int i=link[u].size()-1;i>=0;i--){
46. int v=link[u][i];
47. if(base[u]!=base[v]&&match[u]!=v)
48. if(v==start||(match[v]>=0&&pred[match[v]]>=0))
49. BlossomContract(u,v);
50. else if(pred[v]==-1){
51. pred[v]=u;
52. if(match[v]>=0) push(match[v]);
53. else{ finish=v; return true; }
54. }
55. }
56. }
57. return found;
58. }
59. void AugmentPath(){
60. int u=finish,v,w;
61. while(u>=0){ v=pred[u];w=match[v];match[v]=u;match[u]=v;u=w; }
62. }
63. void FindMaxMatching(){
64. for(int i=0;i<n;++i) match[i]=-1;
65. for(int i=0;i<n;++i) if(match[i]==-1) if(FindAugmentingPath(i)) AugmentPath();
66. }

## SBT

1. struct Size\_Balabced\_Tree{
2. int left[MAXN],right[MAXN],s[MAXN],key[MAXN],root,tt;
3. void clear(){
4. memset(left,0,sizeof(left));
5. memset(right,0,sizeof(right));
6. memset(s,0,sizeof(s));
7. memset(key,0,sizeof(key));
8. root=tt=0;
9. }
10. void gr(int &t){
11. int k=left[t];
12. left[t]=right[k];
13. right[k]=t;
14. s[k]=s[t];
15. s[t]=s[left[t]]+s[right[t]]+1;
16. t=k;
17. }
18. void gl(int &t){
19. int k=right[t];
20. right[t]=left[k];
21. left[k]=t;
22. s[k]=s[t];
23. s[t]=s[left[t]]+s[right[t]]+1;
24. t=k;
25. }
26. void \_insert(int &t,int v){
27. if (!t){
28. t=++tt;
29. key[t]=v;
30. s[t]=1;
31. return ;
32. }
33. ++s[t];
34. if (v<key[t]){
35. \_insert(left[t],v);
36. if (s[left[left[t]]]>s[right[t]]) gr(t);
37. }
38. else{
39. \_insert(right[t],v);
40. if (s[right[right[t]]]>s[left[t]]) gl(t);
41. }
42. }
43. int \_dele(int &t,int v){
44. int ans;
45. --s[t];
46. if (v==key[t] || v<key[t] && !left[t] || v>key[t] && !right[t]){
47. ans=key[t];
48. if (!left[t] || !right[t]) t=left[t]+right[t];
49. else key[t]=\_dele(left[t],v+1);
50. }
51. else
52. if (v<key[t]) ans=\_dele(left[t],v);
53. else ans=\_dele(right[t],v);
54. return ans;
55. }
56. bool \_find(int &t,int v){
57. if (!t) return false;
58. if (key[t]==v) return true;
59. if (v<key[t]) return \_find(left[t],v);
60. else return \_find(right[t],v);
61. }
62. int \_rank(int &t,int v){ //查v的排名
63. if (t==0) return 1;
64. if (v<=key[t]) return \_rank(left[t],v);
65. else return s[left[t]]+1+ \_rank(right[t],v);
66. }
67. int \_selectmintomax(int &t,int k){ //查第k名 (从小到大)
68. if (k==s[left[t]]+1) return key[t];
69. if (k<=s[left[t]]) return \_selectmintomax(left[t],k);
70. else return \_selectmintomax(right[t],k-1-s[left[t]]);
71. }
72. int \_selectmaxtomin(int &t,int k){ //查第k名 从大到小排名
73. if (k==s[right[t]]+1) return key[t];
74. if (k<=s[right[t]]) return \_selectmaxtomin(right[t],k);
75. else return \_selectmaxtomin(left[t],k-1-s[right[t]]);
76. }
77. int \_pred(int &t,int v){ //找比v小的最大的 找不到就返回自己
78. int ans;
79. if (t==0) return v;
80. if (v<=key[t]) ans=\_pred(left[t],v);
81. else{
82. ans=\_pred(right[t],v);
83. if (ans==v) ans=key[t];
84. }
85. return ans;
86. }
87. int \_succ(int &t,int v){ //找比v大的最小的数字 找不到返回自己
88. int ans;
89. if (t==0) return v;
90. if (v>=key[t]) ans=\_succ(right[t],v);
91. else {
92. ans=\_succ(left[t],v);
93. if (ans==v) ans=key[t];
94. }
95. return ans;
96. }
97. void insert(int k){ //插入K
98. \_insert(root,k);
99. }
100. bool find(int k){ //查找K，返回ture false
101. return \_find(root,k);
102. }
103. int dele(int k){ //删除K，如果有就删掉，没有就删掉遍历到的最后一个数字，并且返回这个数字（你可以重新插入）
104. return \_dele(root,k);
105. }
106. int rank(int k){ //查找K在树种的排名，从小到大排名
107. return \_rank(root,k);
108. }
109. int selectmintomax(int k){ //插找第K小元素
110. return \_selectmintomax(root,k);
111. }
112. int selectmaxtomin(int k){ //查找第K大元素
113. return \_selectmaxtomin(root,k);
114. }
115. int pred(int k){ //找比K小的最大的数字
116. return \_pred(root,k);
117. }
118. int succ(int k){ //找比K大的最小的数字
119. return \_succ(root,k);
120. }
121. }SBT;

## Treap

1. //调用方法：
2. //warning:用之前初始化node\_idx
3. //insert(int x); 插入x
4. //remove(int x); 删除x
5. //int kth(int k); 找到第k大的值
6. //int rank(int x); 找到x是第几大
7. //bool find(int x); 判断树里面有没有x
8. struct Node {
9. Node\* ch[2]; //左右子树
10. int r; //优先级，越大越高
11. int v; //值
12. int s;//size 以它为跟的子树的总结点数
13. Node(int v):v(v) {
14. ch[0] = ch[1] = NULL;
15. r = rand();
16. s = 1;
17. }
18. Node(){}
19. bool operator < (const Node& rhs) const {
20. return r < rhs.r;
21. }
22. int cmp(int x) const {
23. if (x == v) return -1;
24. return x < v ? 0 : 1;
25. }
26. void maintain() {
27. s = 1;
28. if (ch[0] != NULL) s += ch[0]->s;
29. if (ch[1] != NULL) s += ch[1]->s;
30. }
31. };
32. Node treap\_node[maxn];//节点数
33. int node\_idx;//用之前初始化 treap\_node的下标
34. struct Treap {
35. Node\* root;
36. Treap():root(NULL) {}
37. /\*五个接口\*/
38. void insert(int x) { insert(root, x); }
39. void remove(int x) { remove(root, x); }
40. int kth(int k) { return kth(root, k); }
41. int rank(int x) { return rank(root, x); }
42. bool find(int x) { return find(root, x); }
43. //d=0代表左旋，d=1代表右旋
44. void rotate(Node\* &o, int d) {//注释左旋为例
45. Node\* k = o->ch[d^1];
46. o->ch[d^1] = k->ch[d]; //o右变成o右左
47. k->ch[d] = o;//o右左变成o
48. o->maintain();
49. k->maintain();
50. o = k;//o变成o右
51. }
52. //在以o为根的子树中插入键值x，修改o
53. void insert(Node\* &o, int x) {
54. if (o == NULL) { //如果根节点为空
55. o = &treap\_node[node\_idx++];
56. \*o = Node(x);
57. return;
58. }
59. int d = (x < o->v ? 0 : 1);//不要用cmp函数，因为可能会有相同结点
60. insert(o->ch[d], x); //x<v?插左:插右
61. if (o->ch[d] ->r > o->r) {//左边优先级大？右旋：左旋
62. rotate(o, d^1);
63. }
64. o->maintain();
65. }
66. void remove(Node\* &o, int x) {
67. int d = o->cmp(x);
68. int ret = 0;
69. if (d == -1) { //找到了
70. Node\* u = o;
71. if (o->ch[0] != NULL && o->ch[1] != NULL) { //如果有两个节点
72. int d2 = (o->ch[0]->r > o->ch[1]->r ? 1 : 0);//如果左结点优先级高则右旋，反之左旋
73. rotate(o, d2);
74. remove(o->ch[d2], x);//右旋后根节点跑到右面所以去右面删除，左旋亦然
75. return;
76. }
77. if (o->ch[0] == NULL) {
78. o = o->ch[1]; //左结点为空，则直接变为右结点
79. }
80. else {
81. o = o->ch[0];//否则右结点为空，则直接变为左结点
82. }
83. delete u;
84. }
85. remove(o->ch[d], x); //递归向下找
86. if (o != NULL) o->maintain();
87. }
88. int kth(Node\* o, int k) {//第k大的值
89. if (o == NULL || k <= 0 || k > o->s) return 0;
90. int s = (o->ch[1] == NULL ? 0 : o->ch[1]->s); //s为右子树结点的个数
91. if (k == s+1) return o->v; //如果k为右子树节点的个数加一，则该节点就是要找的
92. else if (k <= s) return kth(o->ch[1], k); //如果k<=s 即该节点还不够大 继续向右
93. else return kth(o->ch[0], k-s-1);//否则向左走，而且是相对于左边节点的第k-s-1大的，因为去掉了k-s-1
94. }
95. bool find(Node\* o, int x) {
96. while (o != NULL) {
97. int d = o->cmp(x);
98. if (d == -1) return true;
99. else o = o->ch[d];
100. }
101. return false;
102. }
103. int rank(Node\* o, int x) { //从大到小
104. int s = (o->ch[1] == NULL ? 0 : o->ch[1]->s);
105. if (x == o->v) return s + 1;
106. if (x > o->v) return rank(o->ch[1], x);
107. return s + 1 + rank(o->ch[0], x);
108. }
109. } tr;

## Trie字典树

1. //在字典树里面插入一个数的二进制表示
2. //删除一个数的二进制串
3. //与里面的每个数异或的最大值
4. const int maxnode = 35 \* 200000;
5. const int sigmaSize = 2;
6. struct Trie {
7. int ch[maxnode][sigmaSize];//初始化memset(ch[0], 0, sizeof(ch[0]));
8. pii val[maxnode];//表示该节点下面有多少个叶子节点, 层数
9. int cnt[maxnode];//该节点出现的次数
10. int sz;//节点数//初始化sz = 1;
11. char t[40], s[40];
12. void insert(int num) {
13. int len = 35, u = 0;
14. for (int i = 0; i < len; i++) t[i] = ((num >> i) & 1);
15. for (int i = 0; i < len; i++) s[i] = t[len-i-1];
16. for (int i = 0; i < len; i++) {
17. val[u].first++;
18. int c = s[i];
19. if (ch[u][c] < 0) { //节点不存在
20. memset(ch[sz], -1, sizeof(ch[sz]));
21. val[sz].second = val[u].second + 1;
22. ch[u][c] = sz++;//新建节点
23. }
24. cnt[ch[u][c]]++;
25. u = ch[u][c];
26. }
27. val[u].first++;
28. }
29. void remove(int num)
30. {
31. int len = 35, u = 0;
32. for (int i = 0; i < len; i++) t[i] = ((num >> i) & 1);
33. for (int i = 0; i < len; i++) s[i] = t[len-i-1];
34. for (int i = 0; i < len; i++) {
35. val[u].first--;
36. int c = s[i];
37. cnt[ch[u][c]]--;
38. u = ch[u][c];
39. }
40. val[u].first--;
41. }
42. int query(int num)
43. {
44. int len = 35, u = 0;
45. for (int i = 0; i < len; i++) t[i] = ((num >> i) & 1);
46. for (int i = 0; i < len; i++) s[i] = t[len-i-1];
47. int ans = 0;
48. for (int i = 0; i < len; i++)
49. {
50. int c = s[i] ^ 1;
51. int tc = s[i];
52. ans \*= 2;
53. if (ch[u][c] > 0 && cnt[ch[u][c]] > 0) //如果存在就走
54. {
55. ans += 1;
56. u = ch[u][c];
57. continue;
58. }
59. if (ch[u][tc] > 0 && cnt[ch[u][tc]] > 0) //另一边存在
60. {
61. u = ch[u][tc];
62. continue;
63. }
64. }
65. return ans;
66. }
67. };

## Splay

1. struct Node {
2. Node\* c[2];//左右儿子
3. int key;//键
4. int size;//本身及其子树的大小
5. Node() : key(0), size(0) {
6. c[0] = c[1] = this;
7. }
8. Node(int key, Node\* c0, Node\* c1) : key(key) {
9. c[0] = c0;
10. c[1] = c1;
11. }
12. Node\* rz() { //maintain一下，并返回自己节点的指针
13. size = c[0]->size + c[1]->size + 1;
14. return this;
15. }
16. }Tnull, \*null = &Tnull;//null是一颗key=size=0, c[0]=c[1]=this的树的指针
17. struct Splay {
18. Node\* root;
19. Splay() {
20. root = (new Node(\*null))->rz();
21. root->key = INF;
22. }
23. void zig(bool d) {//以0为例(最终结果root指向了他原本左子树指向的内容, null左子树指向一个Node\* A，A的左边指向原来null的左边，A的右边指向原来root的左边
24. Node\* t = root->c[d];//老root的左子树指向的内容给t临时保存
25. root->c[d] = null->c[d];//root的左子树指向null的左子树指向的内容
26. null->c[d] = root;//null的左子树指向root指向的内容
27. root = t;//root指向的内容改为原来左子树的内容
28. }
29. void zigzig(bool d) {//以0为例(最终结果把左左提到了根
30. Node\* t = root->c[d]->c[d];//t指向老root左左指向的内容
31. root->c[d]->c[d] = null->c[d];//
32. null->c[d] = root->c[d];
33. root->c[d] = null->c[d]->c[!d];
34. null->c[d]->c[!d] = root->rz();
35. root = t;
36. }
37. void finish(bool d) {//以0为例（搞不懂了。。。。等会再看
38. Node\* t = null->c[d];//t指向null左指向的内容
39. Node\* p = root->c[!d];//p指向root右指向的内容
40. while (t != null) {
41. t = null->c[d]->c[d];
42. null->c[d]->c[d] = p;
43. p = null->c[d]->rz();
44. null->c[d] = t;
45. }
46. root->c[!d] = p;
47. }
48. void select(int k) {//把左结点的大小变为k(具体细节难啊(T\_T)
49. int t;
50. for (;;) {
51. bool d = k > (t = root->c[0]->size);//t为root左节点的大小
52. if (k == t || root->c[d] == null) break;//
53. if (d) k -= t + 1;//如果k大于左结点的大小 则减去(左结点的大小+1)
54. bool dd = k > (t = root->c[d]->c[0]->size);
55. if (k == t || root->c[d]->c[dd] == null) {
56. zig(d);
57. break;
58. }
59. if (dd) k -= t + 1;
60. d != dd ? zig(d), zig(dd) : zigzig(d);
61. }
62. finish(0), finish(1);
63. root->rz();
64. }
65. void search(int x) {//找到第一个大于或等于x的数，把它splay到root
66. for (;;) {//
67. bool d = x > root->key;//x大向右找，否则向左找
68. if (root->c[d] == null) break;//如果要找的东西不存在，即root->key是仅次于x最大(小)的数
69. bool dd = x > root->c[d]->key;//继续判断下一个节点
70. if (root->c[d]->c[dd] == null) {//如果不存在，就把c[d]转到根，root->又变成了仅次于x最大(小)的数
71. zig(d);
72. break;
73. }
74. d != dd ? zig(d), zig(dd) : zigzig(d);//不同就两个方向转，相同就一个方向转
75. }
76. finish(0), finish(1);
77. root->rz();//root maintain
78. if (x > root->key) select(root->c[0]->size + 1);//如果x大，就把左结点变大一个
79. }
80. void insert(int x) {//插入节点
81. search(x);//先寻找x
82. Node\* oldroot = root;
83. root = new Node(x, oldroot->c[0], oldroot);//root变成新的root（即root的key变成x，左子树不变，右子树变成原来的整棵树
84. oldroot->c[0] = null;//老root的左子树变成null
85. oldroot->rz();//maintain老root
86. root->rz();//maintain新root
87. }
88. void remove(int x) {//删除x，如果x不存在就删除第一个大于x的
89. search(x);//找到x
90. Node \*oldroot = root;
91. root = root->c[1];
92. select(0);//把右边最小的放到root上，左边还是左边
93. root->c[0] = oldroot->c[0];
94. root->rz();//maintain一下
95. delete oldroot;//删掉老树
96. }
97. int kth(int k) {//选择第k大的数
98. select(k - 1);
99. return root->key;
100. }
101. int rank(int x) {//找到x是第几大
102. search(x);
103. return root->c[0]->size + 1;
104. }
105. }sp;

## 并查集

1. int pa[maxn];
2. int find(int x) {
3. return pa[x] != x ? pa[x] = find(pa[x]) : x;
4. }
5. void join(int x, int y) {
6. int fx = find(x), fy = find(y);
7. if(fx != fy) pa[fx] = fy;
8. }
9. /\*
10. int find(int x) { //非递归的带路径压缩
11. int r = x;
12. while (pa[r] != r) r = pa[r];
13. for (int i = x, j; i != r; i = j) { //update
14. j = pa[i]; pa[i] = r;
15. }
16. return r;
17. }
18. \*/
19. //带距离记录的并查集
20. int pa[maxn], d[maxn];
21. int find(int x) {
22. if (pa[x] != x) {
23. int root = find(pa[x]);
24. d[x] += d[pa[x]];
25. return pa[x] = root;
26. }
27. else return x;
28. }

## RMQ问题（Tarjan的Sparse-Table算法）

1. //max or min 自行调整
2. int d[maxn][maxlog];//从i开始,长度为2^j的最值
3. void init(int\* A, int n) { //初始化传入数组 和 大小
4. for (int i = 0; i < n; ++i) d[i][0] = A[i];
5. for (int j = 1; (1<<j) <= n; ++j)
6. for (int i = 0; i + (1<<j) - 1 < n; ++i)
7. d[i][j] = max(d[i][j-1], d[i+(1<<(j-1))][j-1]);
8. }
9. int query(int L, int R) {
10. int k = logz(R-L+1);
11. // int k = 0; while ((1<<(k+1)) <= R-L+1) k++;
12. return max(d[L][k], d[R-(1<<k)+1][k]);
13. }

## 二叉索引树（树状数组）二维

1. #define lowbit(x) (x)&-(x)
2. struct BIT {//Binary Indexed Tree
3. int n, m; //下标从1开始
4. LL C[maxn][maxn];
5. void init(int n, int m) {
6. this->n = n + 5;
7. this->m = m + 5;
8. memset(C, 0, sizeof(C));
9. }
10. LL sum(int x, int y, LL ret = 0) { //x, y右下角的前缀和（含x y）
11. for (; x > 0; x -= lowbit(x))
12. for (int i = y; i > 0; i -= lowbit(i))
13. ret ^= C[x][i];
14. return ret;
15. }
16. void add(int x, int y, LL d) { //点x y 加 d
17. for (; x <= n; x += lowbit(x))
18. for (int i = y; i <= m; i += lowbit(i))
19. C[x][i] ^= d;
20. }
21. }

## 基于二分的离散化

1. const int bg = 0; //下标从0开始
2. struct LS {//离散
3. int e[maxn];
4. int size;
5. void clear() { size = bg; } //所以从bg开始
6. void insert(int k) { e[size++] = k; }
7. void process() {
8. sort(e + bg, e + size);
9. size = unique(e + bg, e + size) - e;
10. }
11. //查找元素k的下标
12. int idx(int k) { return lower\_bound(e + bg, e + size, k) - e; }
13. //查找比K大的下标,k不一定存在
14. int get2(int k) { return upper\_bound(e + bg, e + size, k) - e; }
15. //查找小于等于k的下标,k不一定存在
16. int get3(int k) { return get2(k) - 1; }
17. //返回第x小的元素的值
18. int operator [] (int x) { return e[x]; }
19. } e;

# --------其他：

## 最少找硬币

1. /\*==================================================\*\
2. | 最少找硬币问题（贪心策略-深搜实现）
3. \\*==================================================\*/
4. int value[7] = {100, 50, 20, 10, 5, 2, 1}; //分别有哪些面值
5. int count[7] = {1, 1, 1, 1, 1, 1, 1}; // count[i]:value[i]硬币的个数
6. int res[7];
7. bool flag;
8. void DFS(int total, int p){
9. if(flag) return ;
10. if(p == 7) {
11. if(total == 0) flag = true;
12. return;
13. }
14. int max = total/value[p];
15. if(max > count[p] ) max = count[p];
16. for(int i=max; i >= 0; i--){
17. res[p] = i;
18. DFS(total-i\*value[p], p+1);
19. if( flag ) return ;
20. }
21. }
22. int main() {
23. int pay = 75;
24. flag = false; // 标识是否已经找到结果
25. for(int i=0; i < 7; ++i) res[i] = 0;
26. DFS(pay, 0); // pay为要找的钱数
27. if(flag){
28. printf("Accept\n%d", res[0]);
29. for(int i=1; i < 7; ++i ) printf(" %d", res[i]);
30. printf("\n");
31. }
32. else printf("Refuse\n"); // 无法正好找钱
33. }

## 棋盘分割

1. /\*==================================================\*\
2. | 棋盘分割
3. | 将一个８\*８的棋盘进行如下分割：将原棋盘割下一块矩形棋盘并使剩下部
4. | 分也是矩形，再将剩下的部分继续如此分割，这样割了(n-1)次后，连同最
5. | 后剩下的矩形棋盘共有 n 块矩形棋盘。(每次切割都只能沿着棋盘格子的边
6. | 进行) 原棋盘上每一格有一个分值，一块矩形棋盘的总分为其所含各格分|
7. | 值之和。现在需要把棋盘按上述规则分割成 n 块矩形棋盘，并使各矩形棋
8. ||盘总分的均方差最小。 均方差…，其中平均值…，xi 为第 i 块矩形棋盘的|
9. | 总分。请编程对给出的棋盘及 n，求出 O'的最小值。
10. | POJ 1191 棋盘分割
11. \\*==================================================\*/
12. #define min(a, b) ( (a) < (b) ? (a) : (b) )
13. const int oo = 10000000;
14. int map[8][8];
15. double C[16][8][8][8][8];//c[k][si][ei][sj][ej]: 对矩阵
16. //map[si...sj][ei...ej]分割成 k 个矩形(切割 k-1 刀)的结果
17. double ans; // 平均值
18. int n; // 分成 n 块矩形棋盘
19. void input(void);
20. void reset(void);
21. double caluate(int i1, int j1, int i2, int j2);
22. void dp(int m, int si, int sj, int ei, int ej);
23. int main(void){
24. int m, i, j, k, l;
25. while( scanf("%d", &n) != EOF ){
26. input(); reset();
27. for( m=1; m <= n; m++ ) for( i=0; i < 8; i++ )
28. for( j=0; j < 8; j++ ) for( k=0; k < 8; k++ )
29. for( l=0; l < 8; l++ )
30. if( (k-i+1)\*(l-j+1) < m ) C[m][i][j][k][l] = oo;
31. else if( m == 1 ) C[m][i][j][k][l] = pow( (caluate(i,j,k,l)-ans), 2 );
32. else dp(m, i, j, k, l);
33. printf("%.3lf\n", sqrt(C[n][0][0][7][7]/n));
34. }
35. return 0;
36. }
37. void input(void){
38. int i, j;
39. double sum = 0;
40. for( i=0; i < 8; i++ )
41. for( j=0; j < 8; j++ ){
42. scanf("%d", &map[i][j]);
43. sum += map[i][j];
44. }
45. ans = sum/double(n); // 平均值
46. }
47. void reset(void){
48. int i, j, k, l, m;
49. for(m=0; m <= n; m++) for(i=0; i < 8; i++ )
50. for(j=0; j < 8; j++) for(k=0; k < 8; k++ )
51. for(l=0; l < 8; l++) C[m][i][j][k][l] = 0;
52. }
53. double caluate(int i1, int j1, int i2, int j2){
54. double sum=0;
55. int i, j;
56. for( i=i1; i <= i2; i++ )
57. for( j=j1; j <= j2; j++ ) sum += map[i][j];
58. return sum;
59. }
60. void dp(int m, int si, int sj, int ei, int ej){
61. int i, j;
62. double mins = oo;
63. for( j=sj; j < ej; j++ ) {// 竖刀
64. mins = min(mins, C[1][si][sj][ei][j]+C[m-1][si][j+1][ei][ej]);
65. mins = min(mins, C[m-1][si][sj][ei][j]+C[1][si][j+1][ei][ej]);
66. }
67. for( i=si; i < ei; i++ ) { // 横刀
68. mins = min(mins, C[1][si][sj][i][ej]+C[m-1][i+1][sj][ei][ej]);
69. mins = min(mins, C[m-1][si][sj][i][ej]+C[1][i+1][sj][ei][ej]);
70. }
71. C[m][si][sj][ei][ej] = mins;
72. }

## 汉罗塔

1. /\*==================================================\*\
2. | 汉诺塔
3. | 1,2,...,n 表示 n 个盘子．数字大盘子就大．n 个盘子放在第１根柱子上．大
4. | 盘不能放在小盘上．在第１根柱子上的盘子是 a[1],a[2],...,a[n].
5. |a[1]=n,a[2]=n-1,...,a[n]=1.即 a[1]是最下面的盘子．把 n 个盘子
6. ||移动到第 3 根柱子．每次只能移动 1 个盘子，且大盘不能放在小盘上．问|
7. 第 m 次移动的是哪一个盘子，从哪根柱子移到哪根柱子.例如：n=3,m=2. 回
8. |答是 ：2 1 2，即移动的是 2 号盘，从第 1 根柱子移动到第 2 根柱子 。
9. | HDU 2511 汉诺塔 X
10. 一号柱有 n 个盘子,叫做源柱.移往 3 号柱,叫做目的柱.2 号柱叫做中间柱.
11. 全部移往 3 号柱要 f(n) =（2^n）- 1 次.
12. 最大盘 n 号盘在整个移动过程中只移动一次,n-1 号移动 2 次,i 号盘移动
13. 2^(n-i)次.
14. 1 号盘移动次数最多,每 2 次移动一次.
15. 第 2k+1 次移动的是 1 号盘,且是第 k+1 次移动 1 号盘.
16. 第 4k+2 次移动的是 2 号盘,且是第 k+1 次移动 2 号盘.
17. ......
18. 第(2^s)k+2^(s-1)移动的是 s 号盘,这时 s 号盘已被移动了 k+1 次.
19. 每 2^s 次就有一次是移动 s 号盘.
20. 第一次移动 s 号盘是在第 2^(s-1)次.
21. 第二次移动 s 号盘是在第 2^s+2^(s-1)次.
22. ......
23. 第 k+1 次移动 s 号盘是在第 k\*2^s+2^(s-1)次.
24. 1--2--3--1 叫做顺时针方向,1--3--2--1 叫做逆时针方向.
25. 最大盘 n 号盘只移动一次:1--3,它是逆时针移动.
26. n-1 移动 2 次:1--2--3,是顺时针移动.
27. 如果 n 和 k 奇偶性相同,则 k 号盘按逆时针移动,否则顺时针.
28. \\*==================================================\*/
29. int main() {
30. int i, k;
31. scanf("%d", &k);
32. for( i=0; i < k; i++ ){
33. int n, l;
34. LL m, j, s, t;
35. scanf("%d%lld", &n, &m);
36. s = 1; t = 2;
37. for( l=1; l <= n; l++ ){
38. if( m%t == s ) break;
39. s = t; t \*= 2;
40. }
41. printf("%d ", l);
42. j = m/t;
43. if( n%2 == l%2 ){// 逆时针
44. if( (j+1)%3 == 0 ) printf("2 1\n");
45. if( (j+1)%3 == 1 ) printf("1 3\n");
46. if( (j+1)%3 == 2 ) printf("3 2\n");
47. }
48. else{// 逆时针
49. if( (j+1)%3 == 0 ) printf("3 1\n");
50. if( (j+1)%3 == 1 ) printf("1 2\n");
51. if( (j+1)%3 == 2 ) printf("2 3\n");
52. }
53. }
54. return 0;
55. }

## 求阶乘最后非零位是几

1. //求阶乘最后非零位是几,复杂度 O(nlogn)
2. //返回该位,n 以字符串方式传入
3. int lastdigit(char\* buf){
4. const int mod[20]={1,1,2,6,4,2,2,4,2,8,4,4,8,4,6,8,8,6,8,2};
5. int len=strlen(buf), a[maxn], i, c, ret=1;
6. if (len == 1) return mod[buf[0]-'0'];
7. for (i=0; i<len; i++) a[i]=buf[len-1-i]-'0';
8. for (;len; len-=!a[len-1]) {
9. ret = ret \* mod[a[1]%2\*10+a[0]] % 5;
10. for (c=0,i=len-1; i>=0; i--)
11. c=c\*10+a[i],a[i]=c/5,c%=5;
12. }
13. return ret + ret % 2 \* 5;
14. }

## 约瑟夫环（n个人报数，报道m的出局，求最后一个人）

1. // Joseph's Problem
2. // input: n,m -- the number of persons, the inteval between persons
3. // output: -- return the reference of last person
4. int josephus0(int n, int m)
5. {
6. if (n == 2) return (m%2) ? 2 : 1;
7. int v = (m+josephus0(n-1,m)) % n;
8. if (v == 0) v = n;
9. return v;
10. }
11. int josephus(int n, int m)
12. {
13. if (m == 1) return n;
14. if (n == 1) return 1;
15. if (m >=n) return josephus0(n,m);
16. int l = (n/m)\*m;
17. int j = josephus(n - (n/m), m);
18. if (j <= n-l) return l+j;
19. j -= n-l;
20. int t = (j/(m-1))\*m;
21. if ((j % (m-1)) == 0) return t-1;
22. return t + (j % (m-1));
23. }

## N皇后构造解

1. //构造一组n皇后的解
2. void even1(int n,int \*p) {
3. for (int i=1;i<=n/2;i++)
4. p[i-1]=2\*i;
5. for (int i=n/2+1;i<=n;i++)
6. p[i-1]=2\*i-n-1;
7. }
8. void even2(int n,int \*p){
9. for (int i=1;i<=n/2;i++)
10. p[i-1]=(2\*i+n/2-3)%n+1;
11. for (int i=n/2+1;i<=n;i++)
12. p[i-1]=n-(2\*(n-i+1)+n/2-3)%n;
13. }
14. void generate(int,int\*);
15. void odd(int n,int \*p){ generate(n-1,p),p[n-1]=n; }
16. void generate(int n,int \*p){
17. if (n&1) odd(n,p);
18. else if (n%6!=2) even1(n,p);
19. else even2(n,p);
20. }

## 布尔母函数

1. //判 m[]个价值为 w[]的货币能否构成 value
2. //适合 m[]较大 w[]较小的情况
3. //返回布尔量
4. //传入货币种数 n,个数 m[],价值 w[]和目标值 value
5. #define MAXV 100000
6. int genfunc(int n,int\* m,int\* w,int value){
7. int k,c;
8. char r[MAXV] = {1};
9. for (int i=0;i<n;i++) {
10. for (int j=0;j<w[i];j++){
11. c=m[i]\*r[k=j];
12. while ((k+=w[i])<=value)
13. if (r[k]) c=m[i];
14. else if (c) r[k]=1,c--;
15. if (r[value]) return 1;
16. }
17. }
18. return 0;
19. }

## 取第k小的元素

1. //取第k小的元素,k=0..n-1
2. //平均复杂度 O(n)
3. //注意 a[]中的顺序被改变
4. #define \_cp(a,b) ((a)<(b))
5. typedef int elem\_t;
6. int kth\_element(int n, int\* a, int k) {
7. int l=0, r=n-1, i, j, t, key;
8. while (l<r){
9. for (key=a[((i=l-1)+(j=r+1))>>1];i<j;){
10. for (j--;\_cp(key,a[j]);j--);
11. for (i++;\_cp(a[i],key);i++);
12. if (i<j) t=a[i],a[i]=a[j],a[j]=t;
13. }
14. if (k>j) l=j+1;
15. else r=j;
16. }
17. return a[k];
18. }

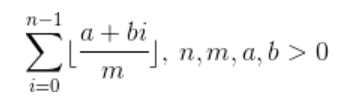
## 幻方构造

1. //幻方构造(l!=2)
2. //横竖斜的和都相等
3. #define MAXN 100
4. void dllb(int l,int si,int sj,int sn,int d[][MAXN]){
5. int n,i=0,j=l/2;
6. for (n=1;n<=l\*l;n++){
7. d[i+si][j+sj]=n+sn;
8. if (n%l){
9. i=(i)?(i-1):(l-1);
10. j=(j==l-1)?0:(j+1);
11. }
12. else i=(i==l-1)?0:(i+1);
13. }
14. }
15. void magic\_odd(int l,int d[][MAXN]) {dllb(l,0,0,0,d);}
16. void magic\_4k(int l,int d[][MAXN]){
17. int i,j;
18. for (i=0;i<l;i++) for (j=0;j<l;j++)
19. d[i][j]=((i%4==0||i%4==3)&&(j%4==0||j%4==3)||(i%4==1||i%4==2)&&(j%4==1||j%4==2))
20. ?(l\*l-(i\*l+j)):(i\*l+j+1);
21. }
22. void magic\_other(int l,int d[][MAXN]){
23. int i,j,t;
24. dllb(l/2,0,0,0,d);
25. dllb(l/2,l/2,l/2,l\*l/4,d);
26. dllb(l/2,0,l/2,l\*l/2,d);
27. dllb(l/2,l/2,0,l\*l/4\*3,d);
28. for (i=0;i<l/2;i++) for (j=0;j<l/4;j++) if (i!=l/4||j)
29. t=d[i][j],d[i][j]=d[i+l/2][j],d[i+l/2][j]=t;
30. t=d[l/4][l/4],d[l/4][l/4]=d[l/4+l/2][l/4],d[l/4+l/2][l/4]=t;
31. for (i=0;i<l/2;i++) for (j=l-l/4+1;j<l;j++)
32. t=d[i][j],d[i][j]=d[i+l/2][j],d[i+l/2][j]=t;
33. }
34. void generate(int l,int d[][MAXN]){
35. if (l%2) magic\_odd(l,d);
36. else if (l%4==0) magic\_4k(l,d);
37. else magic\_other(l,d);
38. }

## 蔡勒（Zeller）公式

1. 是一个计算星期的公式，随便给一个日期，就能用这个公式推算出是星期几。
2. int zeller(int y,int m,int d) {
3. if (m<=2) y--,m+=12; int c=y/100; y%=100;
4. int w=((c>>2)-(c<<1)+y+(y>>2)+(13\*(m+1)/5)+d-1)%7;
5. if (w<0) w+=7; return(w);
6. }

## 直线下格点统计



1. LL solve(LL n,LL a,LL b,LL m){
2. if(b==0) return n\*(a/m);
3. if(a>=m) return n\*(a/m)+solve(n,a%m,b,m);
4. if(b>=m) return (n-1)\*n/2\*(b/m)+solve(n,a,b%m,m);
5. return solve((a+b\*n)/m,(a+b\*n)%m,m,b);
6. }

## 字符串的最小表示

1. string find(string s) {
2. int i,j,k,l,N=s.length(); s+=s;
3. for(i=0,j=1;j<N;){
4. for(k=0;k<N&&s[i+k]==s[j+k];k++);
5. if(k>=N) break;
6. if(s[i+k]<s[j+k]) j+=k+1;
7. else l=i+k,i=j,j=max(l,j)+1;
8. }
9. return s.substr(i,N);
10. }

# -----------大数与分数：

## 分数的相关计算

1. //分数的相关计算
2. struct Frac { int num,den; };
3. int gcd(int a,int b) {
4. int t;
5. if (a<0) a=-a;
6. if (b<0) b=-b;
7. if (!b) return a;
8. while (t=a%b) a=b,b=t;
9. return b;
10. }
11. void simplify(Frac& f) {
12. int t;
13. if (t=gcd(f.num,f.den))
14. f.num/=t,f.den/=t;
15. else f.den=1;
16. }
17. Frac f(int n,int d,int s=1) {
18. Frac ret;
19. if (d<0) ret.num=-n,ret.den=-d;
20. else ret.num=n,ret.den=d;
21. if (s) simplify(ret);
22. return ret;
23. }
24. Frac convert(double x) {
25. Frac ret;
26. for (ret.den=1;fabs(x-int(x))>1e-10;ret.den\*=10,x\*=10);
27. ret.num=(int)x;
28. simplify(ret);
29. return ret;
30. }
31. int fraqcmp(Frac a, Frac b) {
32. int g1=gcd(a.den,b.den),g2=gcd(a.num,b.num);
33. if (!g1||!g2) return 0;
34. return b.den/g1\*(a.num/g2)-a.den/g1\*(b.num/g2);
35. }
36. Frac add(Frac a, Frac b) {
37. int g1=gcd(a.den,b.den),g2,t;
38. if (!g1) return f(1,0,0);
39. t=b.den/g1\*a.num+a.den/g1\*b.num;
40. g2=gcd(g1,t);
41. return f(t/g2,a.den/g1\*(b.den/g2),0);
42. }
43. Frac sub(Frac a, Frac b) {
44. return add(a,f(-b.num,b.den,0));
45. }
46. Frac mul(Frac a, Frac b) {
47. int t1=gcd(a.den,b.num),t2=gcd(a.num,b.den);
48. if (!t1||!t2) return f(1,1,0);
49. return f(a.num/t2\*(b.num/t1),a.den/t1\*(b.den/t2),0);
50. }
51. Frac div(Frac a, Frac b) {
52. return mul(a,f(b.den,b.num,0));
53. }

### 大整数（一）

1. #include <cstdio>
2. #include <cstring>
3. const int base = 10000; // (base^2) fit into int
4. const int width = 4; // width = log base
5. const int N = 1000; // n \* width: 可表示的大位数
6. struct bint {
7. int ln, v[N];
8. bint (int r = 0) { // r应该是字符串！
9. for (ln = 0; r > 0; r /= base) v[ln++] = r % base;
10. }
11. bint& operator = (const bint& r) {
12. memcpy(this, &r, (r.ln + 1) \* sizeof(int));// !
13. return \*this;
14. }
15. };
16. bool operator < (const bint& a, const bint& b) {
17. int i;
18. if (a.ln != b.ln) return a.ln < b.ln;
19. for (i = a.ln - 1; i >= 0 && a.v[i] == b.v[i]; i--);
20. return i < 0 ? 0 : a.v[i] < b.v[i];
21. }
22. bool operator <= (const bint& a, const bint& b) {
23. return !(b < a);
24. }
25. bint operator + (const bint& a, const bint& b) {
26. bint res;
27. int i, cy = 0;
28. for (i = 0; i < a.ln || i < b.ln || cy > 0; i++) {
29. if (i < a.ln) cy += a.v[i];
30. if (i < b.ln) cy += b.v[i];
31. res.v[i] = cy % base; cy /= base;
32. }
33. res.ln = i;
34. return res;
35. }
36. bint operator - (const bint& a, const bint& b) {
37. bint res; int i, cy = 0;
38. for (res.ln = a.ln, i = 0; i < res.ln; i++) {
39. res.v[i] = a.v[i] - cy;
40. if (i < b.ln) res.v[i] -= b.v[i];
41. if (res.v[i] < 0) cy = 1, res.v[i] += base;
42. else cy = 0;
43. }
44. while (res.ln > 0 && res.v[res.ln - 1] == 0) res.ln--;
45. return res;
46. }
47. bint operator \* (const bint& a, const bint& b) {
48. bint res; res.ln = 0;
49. if (0 == b.ln) { res.v[0] = 0; return res; }
50. int i, j, cy;
51. for (i = 0; i < a.ln; i++) {
52. for (j=cy=0; j < b.ln || cy > 0; j++, cy/= base) {
53. if (j < b.ln) cy += a.v[i] \* b.v[j];
54. if (i + j < res.ln) cy += res.v[i + j];
55. if (i + j >= res.ln) res.v[res.ln++] = cy % base;
56. else res.v[i + j] = cy % base;
57. }
58. }
59. return res;
60. }
61. bint operator / (const bint& a, const bint& b) { // ! b != 0
62. bint tmp, mod, res;
63. int i, lf, rg, mid;
64. mod.v[0] = mod.ln = 0;
65. for (i = a.ln - 1; i >= 0; i--) {
66. mod = mod \* base + a.v[i];
67. for (lf = 0, rg = base -1; lf < rg; ) {
68. mid = (lf + rg + 1) / 2;
69. if (b \* mid <= mod) lf = mid;
70. else rg = mid - 1;
71. }
72. res.v[i] = lf;
73. mod = mod - b \* lf;
74. }
75. res.ln = a.ln;
76. while (res.ln > 0 && res.v[res.ln - 1] == 0) res.ln--;
77. return res; // return mod 就是%运算
78. }
79. int digits(bint& a) { // 返回位数
80. if (a.ln == 0) return 0;
81. int l = ( a.ln - 1 ) \* 4;
82. for (int t = a.v[a.ln - 1]; t; ++l, t/=10) ;
83. return l;
84. }
85. bool read(bint& b, char buf[]) { // 读取失败返回0
86. if (1 != scanf("%s", buf)) return 0;
87. int w, u, ln = strlen(buf);
88. memset(&b, 0, sizeof(bint));
89. if ('0' == buf[0] && 0 == buf[1]) return 1;
90. for (w = 1, u = 0; ln; ) {
91. u += (buf[--ln] - '0') \* w;
92. if (w \* 10 == base) {
93. b.v[b.ln++] = u; u = 0; w = 1;
94. }
95. else w \*= 10;
96. }
97. if (w != 1) b.v[b.ln++] = u;
98. return 1;
99. }
100. void write(const bint& v) {
101. int i;
102. printf("%d", v.ln == 0 ? 0 : v.v[v.ln - 1]);
103. for (i = v.ln - 2; i >= 0; i--)
104. printf("%04d", v.v[i]); // ! 4 == width
105. printf("\n");
106. }
107. int main() {}

### 大整数（二）

1. const int maxn = 5000;
2. class BigInt {
3. private:
4. int bit[5000];
5. bool negative;//负数标志
6. public:
7. BigInt(); //默认构造函数，值为0
8. BigInt(const int); //构造函数
9. BigInt(const char \*); //构造函数
10. BigInt(const BigInt &); //复制构造函数
11. /\*重载赋值运算符\*/
12. BigInt& operator=(const BigInt&);
13. BigInt& operator=(const int );
14. /\*重载算数运算符\*/
15. BigInt operator+(const BigInt& )const;
16. BigInt operator+(const int )const;
17. BigInt operator-(const BigInt& )const;
18. BigInt operator-(const int )const;
19. BigInt operator\*(const BigInt& )const;
20. BigInt operator\*(const int )const;
21. BigInt operator/(const int )const;
22. int operator%(const int )const;
23. /\*重载比较运算符\*/
24. bool operator>(const BigInt& )const;
25. bool operator>(const int )const;
26. bool operator>=(const BigInt& )const;
27. bool operator>=(const int )const;
28. bool operator<(const BigInt& )const;
29. bool operator<(const int )const;
30. bool operator<=(const BigInt& )const;
31. bool operator<=(const int )const;
32. bool operator==(const BigInt& )const;
33. bool operator==(const int )const;
34. bool operator!=(const BigInt& )const;
35. bool operator!=(const int )const;
36. void print() const;///输出数值
37. bool isZero() const;///是否为0
38. bool isPositive() const;///是否为正数
39. bool isNegative() const;///是否为负数
40. bool nonNegative() const;///是否为非负数
41. private:
42. BigInt opposite()const;///取相反数
43. BigInt absoluteAdd(const BigInt&)const;///加上绝对值
44. BigInt absoluteMinus(const BigInt&)const;///减去绝对值小于自身的数的绝对值
45. bool absoluteEqual(const BigInt&)const;///绝对值等于
46. bool absoluteGreater(const BigInt&)const;///绝对值大于
47. bool absoluteEqualGreater(const BigInt&)const;///绝对值大于等于
48. };
49. BigInt::BigInt()
50. {
51. memset(bit,0,sizeof(bit));
52. negative = false;
53. }
54. BigInt::BigInt(const int n)
55. {
56. memset(bit,0,sizeof(bit));
57. int nn = n;
58. if (nn>=0) negative = false;
59. else {
60. negative = true;
61. nn = -nn;
62. }
63. int pos = 0;
64. while (nn) {
65. bit[pos++] = nn % 10;
66. nn /= 10;
67. }
68. }
69. BigInt::BigInt(const char \*s)
70. {
71. int len = strlen(s);
72. bool valid = true;//符合数字格式
73. if (len >= 2) {
74. if (s[0]!='+' && s[0]!='-' && !isdigit(s[0])) valid = false;
75. for (int i=1; i<len; ++i) {
76. if (!isdigit(s[i])) valid = false;
77. }
78. }
79. else if (len == 1) {
80. if (!isdigit(s[0])) valid = false;
81. }
82. if (len==0 || !valid) {
83. memset(bit,0,sizeof(bit));
84. negative = false;
85. return;
86. }
87. int beg = 0, end = len-1;
88. if (s[0] == '+') {
89. negative = false;
90. ++beg;
91. }
92. else if (s[0] == '-') {
93. bool zeroFlag = true;
94. for (int i=1; i<len; ++i) {
95. if (s[i]!='0') {
96. zeroFlag = false;
97. break;
98. }
99. }
100. if (zeroFlag) negative = false;
101. else negative = true;
102. ++beg;
103. }
104. else negative = false;
105. memset(bit,0,sizeof(bit));
106. for (int i=beg; i<=end; ++i) {
107. bit[len-1-i] = s[i] - '0';
108. }
109. }
110. BigInt::BigInt(const BigInt& n)
111. {
112. memcpy(bit,n.bit,sizeof(bit));
113. negative = n.negative;
114. }
115. BigInt& BigInt::operator=(const BigInt& n)
116. {
117. memcpy(bit,n.bit,sizeof(bit));
118. negative = n.negative;
119. return \*this;
120. }
121. BigInt& BigInt::operator=(const int n)
122. {
123. return \*this = BigInt(n);
124. }
125. BigInt BigInt::operator+(const BigInt& n)const
126. {
127. if ((!negative && !n.negative) || (negative && n.negative)) {
128. return this->absoluteAdd(n);
129. }
130. else {
131. if (absoluteEqual(n)) return BigInt();
132. else if (absoluteEqualGreater(n)) return this->absoluteMinus(n);
133. else return n.absoluteMinus(\*this);
134. }
135. }
136. BigInt BigInt::operator+(const int n)const
137. {
138. return \*this + BigInt(n);
139. }
140. BigInt BigInt::operator-(const BigInt& n)const
141. {
142. return \*this + n.opposite();
143. }
144. BigInt BigInt::operator-(const int n)const
145. {
146. return \*this - BigInt(n);
147. }
148. BigInt BigInt::operator\*(const BigInt& n)const
149. {
150. if (isZero() || n.isZero()) return BigInt();
151. BigInt bi = BigInt();
152. if ((!negative && !n.negative) || (negative && n.negative)) {
153. bi.negative = false;
154. }
155. else bi.negative = true;
156. for (int i=0; i<maxn; ++i) for (int j=0; j<maxn-i; ++j) {
157. bi.bit[i+j] += bit[i] \* n.bit[j];
158. }
159. for (int i=0; i<maxn-1; ++i) {//进位
160. bi.bit[i+1] += bi.bit[i] / 10;
161. bi.bit[i] %= 10;
162. }
163. return bi;
164. }
165. BigInt BigInt::operator\*(const int n)const
166. {
167. return \*this \* BigInt(n);
168. }
169. BigInt BigInt::operator/(const int n)const
170. {//除以0直接返回0
171. if (isZero() || n==0) return BigInt();
172. BigInt bi = BigInt();
173. if ((!negative && n>0) || (negative && n<0)) {
174. bi.negative = false;
175. }
176. else bi.negative = true;
177. int div = 0;//累计除数
178. for (int i=maxn-1; i>=0; --i) {
179. div = div \* 10 + bit[i];
180. bi.bit[i] = div / n;
181. div %= n;
182. }
183. return bi;
184. }
185. int BigInt::operator%(const int n)const
186. {
187. int mod = 0;//累计余数
188. for (int i=maxn-1; i>=0; --i) {
189. //mod = ((mod\*(MAXBIT+1/\*??\*/)) + bit[i]) % n;
190. mod = ((mod\*10) + bit[i]) % n;
191. }
192. return mod;
193. }
194. bool BigInt::operator>(const BigInt& n)const
195. {
196. if (!negative && n.negative) return true;
197. else if (negative && !n.negative) return false;
198. else if (!negative && !n.negative) return absoluteGreater(n);
199. else return n.absoluteGreater(\*this);
200. }
201. bool BigInt::operator>(const int n)const
202. {
203. return \*this > BigInt(n);
204. }
205. bool BigInt::operator>=(const BigInt& n)const
206. {
207. if (!negative && n.negative) return true;
208. else if (negative && !n.negative) return false;
209. else if (!negative && !n.negative) return absoluteEqualGreater(n);
210. else return n.absoluteEqualGreater(\*this);
211. }
212. bool BigInt::operator>=(const int n)const
213. {
214. return \*this >= BigInt(n);
215. }
216. bool BigInt::operator<(const BigInt& n)const
217. {
218. return n > \*this;
219. }
220. bool BigInt::operator<(const int n)const
221. {
222. return \*this < BigInt(n);
223. }
224. bool BigInt::operator<=(const BigInt& n)const
225. {
226. return n >= \*this;
227. }
228. bool BigInt::operator<=(const int n)const
229. {
230. return \*this <= BigInt(n);
231. }
232. bool BigInt::operator==(const BigInt& n)const
233. {
234. if (negative != n.negative) return false;
235. for (int i=0; i<maxn; ++i) {
236. if (bit[i] != n.bit[i]) return false;
237. }
238. return true;
239. }
240. bool BigInt::operator==(const int n)const
241. {
242. return \*this == BigInt(n);
243. }
244. bool BigInt::operator!=(const BigInt& n)const
245. {
246. if (negative != n.negative) return true;
247. for (int i=0; i<maxn; ++i) {
248. if (bit[i] != n.bit[i]) return true;
249. }
250. return false;
251. }
252. bool BigInt::operator!=(const int n)const
253. {
254. return \*this != BigInt(n);
255. }
256. void BigInt::print()const
257. {
258. if (negative) printf("-");
259. int pos = maxn - 1;
260. for (; pos>0; --pos) {
261. if (bit[pos]) break;
262. }
263. for (int i=pos; i>=0; --i) printf("%d",bit[i]);
264. }
265. bool BigInt::isZero()const
266. {
267. bool zeroFlag = true;
268. for (int i=0; i<maxn; ++i) {
269. if (bit[i] != 0) {
270. zeroFlag = false;
271. break;
272. }
273. }
274. return zeroFlag;
275. }
276. bool BigInt::isPositive()const
277. {
278. return !negative && !isZero();
279. }
280. bool BigInt::isNegative()const
281. {
282. return negative;
283. }
284. bool BigInt::nonNegative()const
285. {
286. return !negative;
287. }
288. BigInt BigInt::opposite()const
289. {
290. BigInt n(\*this);
291. if (!n.isZero()) n.negative = !n.negative;
292. return n;
293. }
294. BigInt BigInt::absoluteAdd(const BigInt& n)const
295. {
296. BigInt bi(\*this);
297. int next = 0;//进位
298. for (int i=0; i<maxn; ++i) {
299. bi.bit[i] = (bit[i] + n.bit[i] + next) % 10;
300. next = (bit[i] + n.bit[i] + next) / 10;
301. }
302. return bi;
303. }
304. BigInt BigInt::absoluteMinus(const BigInt& n)const
305. {
306. BigInt bi(\*this);
307. for (int i=maxn-1; i>=0; --i) {
308. if (bi.bit[i]>=n.bit[i]) bi.bit[i] -= n.bit[i];
309. else {///借位
310. int borrow = i + 1;///借位位
311. while (bi.bit[borrow]==0) ++borrow;
312. --bi.bit[borrow];
313. for (int j=i+1; j<borrow; ++j) bi.bit[j] = 9;
314. bi.bit[i] = bi.bit[i] + 10 - n.bit[i];
315. }
316. }
317. return bi;
318. }
319. bool BigInt::absoluteEqual(const BigInt& n)const
320. {
321. for (int i=0; i<maxn; ++i) {
322. if (bit[i] != n.bit[i]) return false;
323. }
324. return true;
325. }
326. bool BigInt::absoluteGreater(const BigInt& n)const
327. {
328. for (int i=maxn-1; i>=0; --i) {
329. if (bit[i]>n.bit[i]) return true;
330. else if (bit[i]<n.bit[i]) return false;
331. }
332. return false;
333. }
334. bool BigInt::absoluteEqualGreater(const BigInt& n)const
335. {
336. for (int i=maxn-1; i>=0; --i) {
337. if (bit[i]>n.bit[i]) return true;
338. else if (bit[i]<n.bit[i]) return false;
339. }
340. return true;
341. }
342. void solve()
343. {
344. char s1[5000];
345. int s2;
346. cin>>s1;
347. BigInt a(s1);///赋值给大数类
348. cin>>s2;
349. a = a/s2;
350. a.print();
351. }
352. int main()
353. {
354. solve();
355. return 0;
356. }
357. ///支持大数的4个运算，以及负数，但是不支持大数之间的除法，大树与longlong之间的除法支持

# ----------一些常见套路：

## 2 台机器工作调度

2台机器, n件任务, 必须先在S1上做, 再在S2上做. 任务之间先做后

做任意. 求最早的完工时间. 这是一个经典问题: 2台机器的情况下有多

项式算法(Johnson算法), 3台或以上的机器是NP-hard的. Johnson算法:

(1) 把作业按工序加工时间分成两个子集,

第一个集合中在S1上做的时间比在S2上少,

其它的作业放到第二个集合.

先完成第一个集合里面的作业, 再完成第二个集合里的作业.

(2) 对于第一个集合, 其中的作业顺序是按在S1上的时间的不减排列;

对于第二个集合, 其中的作业顺序是按在S2上的时间的不增排列.

## 其他公式定理：

容斥原理

|A∪B∪C| = |A|+|B|+|C|-|A∩B|-|B∩C|-|C∩A|+|A∩B∩C|

k个元素共n个，第i个元素有ni个，全排列个数x为

n1!n2!n3!…nk!x = n!

n个不同元素可重复选，选k个，共有

C(k+n-1, n-1)中选法

二项式定理

D:\Documents\My Knowledge\temp\f485076a-1455-46b1-a1e5-70bfb5590b58\4\index_files\clip_image001d285e21b-4ce1-4a72-996b-a145b4ad494a.png

条件概论公式

P(A|B) = P(AB)/P(B)

P(A|B) = P(B|A)\*P(A)/P(B)

全概率公式

P(A) = P(A|B1¬)\*P(B1)+P(A|B2)\*P(B2)+…+P(A|Bn)\*P(Bn)

期望的线性性质 E(X+Y) = EX+EY

平面图欧拉公式

V – E + F = 2，（V顶点数，E边数，F面数）

1+4+9+16+……+n^2 = n(n+1)(2n+1)/6

1+8+27+64+……n^3 = [n(n+1)/2]²

等差数列求和公式:

D:\Documents\My Knowledge\temp\f485076a-1455-46b1-a1e5-70bfb5590b58\4\index_files\ad1cc7b7-022d-411e-a130-644438129505.png

等比数列求和公式：

D:\Documents\My Knowledge\temp\f485076a-1455-46b1-a1e5-70bfb5590b58\4\index_files\cfd5f01c-1aa0-422b-b2fd-d6539a6e887b.png

任意点(x,y)，绕一个坐标点(rx0,ry0)逆时针旋转a角度后的新的坐标设为(x0, y0)，有公式：

1. x0 = (x - rx0) \* cos(a) - (y - ry0) \* sin(a) + rx0 ;
2. y0 = (x - rx0) \* sin(a) + (y - ry0) \* cos(a) + ry0 ;

定理&公式：---------------------------------------------------------------------------------------------------

1、梅涅劳斯定理

当直线交

D:\Documents\My Knowledge\temp\822aac4b-892e-46b3-916e-088280bfd30f\128\index_files\0.05692620691843331.png

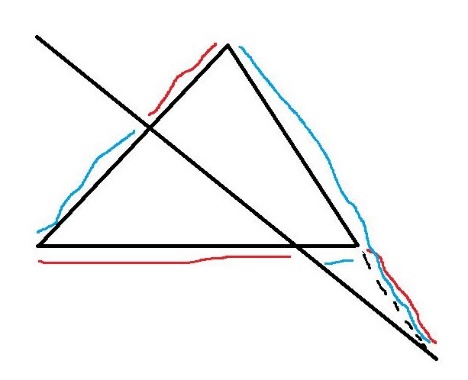
三边所在直线

D:\Documents\My Knowledge\temp\822aac4b-892e-46b3-916e-088280bfd30f\128\index_files\0.2649312422145158.png

于点

D:\Documents\My Knowledge\temp\822aac4b-892e-46b3-916e-088280bfd30f\128\index_files\0.5790828668978065.png

时，D:\Documents\My Knowledge\temp\822aac4b-892e-46b3-916e-088280bfd30f\128\index_files\0.723824905231595.png



2、欧拉定理

设平面图的顶点数、边数和面数分别为V，E和F，则V+F-E=2（面包括封闭区域和无限大区域）

3、皮克定理

一个计算点阵中顶点在格点上的多边形面积公式：S=a+b÷2-1，其中a表示多边形内部的点数，b表示多边形边界上的点数，s表示多边形的面积。

4、知道两点坐标(x1, y1)，(x2, y2)求一般式

Ax+By+C = 0

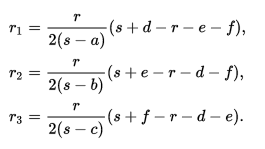
A = (y2 - y1)

B = (x1 - x2)

C = - A \* x1 - B \* y1

5、三角形内部 相互外切的三个圆的半径（Malfatti Circles）

三角形的三边分别为*a*, *b*, *c* 且内切圆半径为*r*, 半周长为s = (a + b + c)/2, *d*, *e*, *f* 分别为内切圆圆心到角ABC的距离，则三个元的半径分别为



经验：----------------------------------------------------------------------------

经验一

甲从A出发，速度为va(向量)，乙从B出发，速度为vb(向量)，

则每一时刻甲乙之间的距离 相当于甲在A点不动，乙从B出发，速度为(vb-va)

经验二

给出平面上n个点，找一条直线，使得所有的点在直线的同侧（也可以在直线上），且到直线的距离之和最小，这条直线一定为这n个点所组成的凸包的边所在的直线。

经验三

一个凸多边形向外扩展L（扩展后的边上的任意点到原凸多边形的距离为L）后的周长为 原图形的周长+2\*PI\*L

经验四

给出多边形没告诉方向时，只需要判断最左边（相同取下）的点的前后的叉积

((P[(min\_idx+1)%n] - P[min\_idx]) ^ (P[(n+min\_idx-1)%n] - P[min\_idx])) < 0为顺时针

# 部分知识点详解

## 暴力的技巧

### 一、01背包

当背包大小数字特别大的时候，只能暴力了。

暴力方法

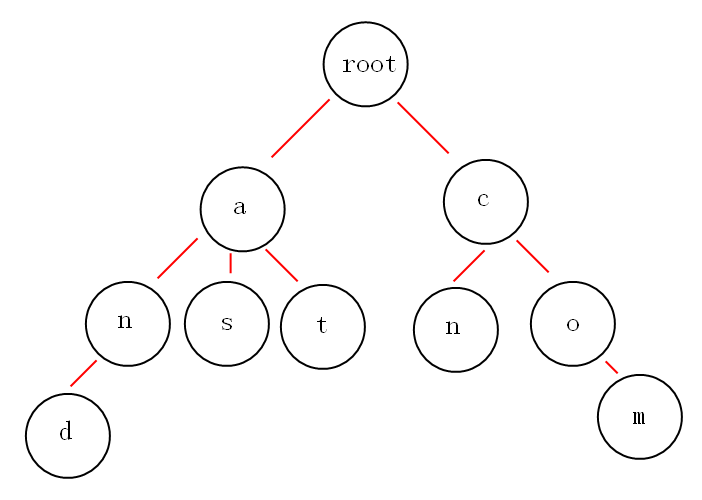
1. 首先给每个物品的性价比排序，从高到低排序
2. 选与不选的方法进行2^n的搜索
3. 剪枝：如果对于当前状态，搜到第k个物品，那么因为性价比已经从高到低排序好，那么我们可以一直选后面的所有物品，对于某个物品，因为体积过大无法放入背包的情况，我们可以把这个物品拆解，用它的“性价比”来填充背包的剩余空间。 如果这种方法不能得到比当前最优解更优的解，则直接进行回溯即可。

### 二、DLX详见DLX相关文件

## AC自动机

### 字母树。

AC自动机，首先需要构建字母树，所谓字母树trie。



这样的字母树中，如果插入一个单词，如果这个单词的前缀曾经出现过了，肯定会节约空间，不需要额外创造空间。

代码

1. const int SIGMA\_SIZE = 26; //字典树分叉数，也就是字母集大小
2. const int MAXNODE = 56789;//字典树总节点个数
3. int ch[MAXNODE][SIGMA\_SIZE];//字典树节点
4. int val[MAXNODE]; // 每个字符串的结尾的特殊记录值
5. int sz; //字典树当前节点个数
6. int match[MAXNODE];//表示字典树中，下标为i的点，是否为XX
7. void init() {//初始化函数
8. sz = 1;
9. memset(ch[0], 0, sizeof(ch[0]));//字典树根节点，每个分叉都指向根
10. memset(val, 0, sizeof(val));//字典树所有节点，的特殊记录值初始化
11. memset(match, 0, sizeof(match));//额外保存的字典树特殊记录值
12. }
13. int idx(char c) // 字符c的编号，为了对应字典树的每个分叉
14. {
15. /\*
16. //if (c == '\0') return 62; //如果包含26字母大小写+数字+\0符号，一共63个
17. //包含所有大小写字母和数字idx函数
18. if (c >= '0' && c <= '9') return c - '0';
19. if (c >= 'a' && c <= 'z') return c - 'a' + 10;
20. return c - 'A' + 36;
21. \*/
22. //return (int)c-'A';
23. return (int)(c-'a');
24. }
25. void insert(char s[], int len, int id) { //对于s[]数组，插入长度为len的字符串，并且这个字符串有特殊含义id
26. int now = 0;
27. for(int i = 0; i < len; i++) {
28. int c = idx(s[i]);
29. if(!ch[now][c]) {
30. memset(ch[sz], 0, sizeof(ch[sz]));
31. val[sz] = 0;
32. ch[now][c] = sz++;
33. }
34. now = ch[now][c];
35. }
36. val[now] = id; //在字符串末尾带入特殊含义，通常直接val[now]=1,表示这里有单词结尾
37. }

上述代码实现了字典树的初始化，和插入一个长为len单词s[]，并且包含id的标记信息。id不一定是必须的，有时候val数组在单词结尾，仅仅记录了这个单词出现的次数，但是有时候又要记录这个单词对应的其他信息，这就需要记录下来。

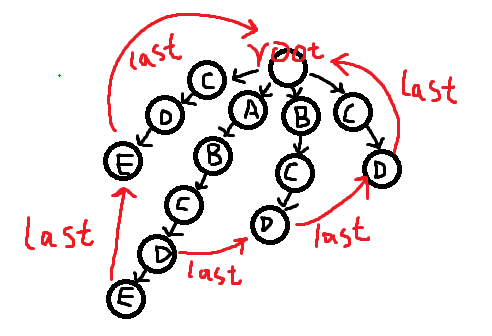
### AC自动机

1、last和f数组介绍

AC自动机可以实现多串匹配功能，初始化时，需要更多的信息。

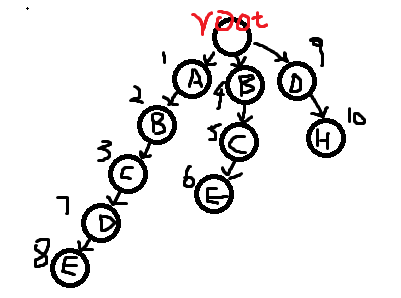
1. int ch[MAXNODE][SIGMA\_SIZE];
2. int f[MAXNODE]; // fail函数
3. int val[MAXNODE];
4. int last[MAXNODE]; // 输出链表的下一个结点
5. int sz;
6. int match[MAXNODE];
7. queue<int>q;
8. void init() {//初始化函数
9. sz = 1;
10. memset(ch[0], 0, sizeof(ch[0]));
11. memset(val, 0, sizeof(val));
12. memset(match, 0, sizeof(match));
13. }

额外添加了last和f数组，以及一个queue。其中，queue是为了之后做BFS提前申明好的变量，last则有有趣的意思。



如上图所示，我们有多个匹配串分别为CDE,ABCDE,BCD,CD。有一个匹配串ABCDE，在trie上“跑”的时候，跑到了图上ABCD的D的位置的时候，实际上已经包含了BCD,和CD。在ABCDE的时候，不仅包含了ABCDE还包含了CDE这个串。所以我们使用last指针，只要last指针不为空（指向0，当然0也是root），那么就说明有匹配串可以一直匹配下去。

对于f也就是fail数组（指针），则表明匹配失败后应该怎么办。令g[i,j]表示从i到j这一路遍历的所有字符串，比如g[1,5]就是ABCDE，g[6,8]就是CDE。总之，g[I,j]就是表示从下标i到下标j这一路上的字符串。并不一定是连续的下标如图所示



这时候g[1,8]表示的是ABCDE。

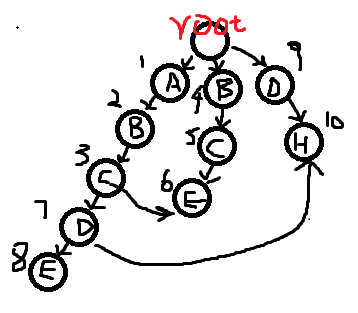
f[i]的意义就是g[?,i]和g[0,f[i]]的字符串是相等的。

比如，f[3]=5,因为g[2,3]与g[root,5]是相同的。再比如f[7]=9,因为g[root,9]与g[7,7]是相同的

last[i] 则表示g[0,last[i]]的字符串，是确定存在的，并且以last[i]结尾的字符串。

2、Trie图

事实上，根据f[]数组，我们可以直接把f的信息反映到ch数组里去，比如上图中，假如有字符串ABCDH，那么匹配顺序是1237，然后这时候匹配失败了，原来需要访问f数组来跳转到9，从9开始匹配字符“H”，但是可以直接在7的 ch[7][“H”]=10,直接处理出来即可。这样，整个图就变为一个Trie图了。



这样的话，遍历起来的程序也方便的多。

3、find函数简介

代码为：

1. void find(char text[], int len) {
2. int j = 0; // 当前结点编号，初始为根结点
3. for(int i = 0; i < len; i++) { // 文本串当前指针
4. int c = idx(text[i]);
5. j = ch[j][c];
6. if(val[j]) print(j);
7. else if(last[j]) print(last[j]); // 找到了！
8. }
9. }

对于长度为len的text，开始跑trie图。依旧以上面的Trie图为例，如果一个text为ABCDEHABCEDH，那么访问为1 2 3 7 8 root 1 2 3 6 9 10。其中有6直接到9，因为对于E要访问D的话，E的f[6]=0，而ch[0][‘D’]=9,所以ch[6][‘D’]=9，则这样直接跑了过去。

除了跑trie图外，find函数还会对于val有特定值（常见的，val一旦为1，表明这里有单词结束。比如对于上面的trie图，val[8],val[6],val[10]记录为1.）然后对特定的val值，或者有last指针存在的情况下，进行print函数操作。

4、print函数简介

代码为：

1. void print(int j) {
2. if(j) {
3. vis[j]=1;
4. print(last[j]);
5. }
6. }

print(j)的意思，就是递归的处理以节点j结尾的所有字符串。只要j不是根节点，那么就进行处理，比如以j为结尾的单词被访问过了。

### AC自动机常见套路

1. 直接的字符串匹配。

裸跑AC自动机即可。

1. void insert(char s[], int len) {
2. int now = 0;
3. for(int i = 0; i < len; i++) {
4. int c = idx(s[i]);
5. if(!ch[now][c]) {
6. memset(ch[sz], 0, sizeof(ch[sz]));
7. val[sz] = 0;
8. ch[now][c] = sz++;
9. }
10. now = ch[now][c];
11. }
12. val[now] = 1;//表示单词在now出现过
13. }

然后find函数不变，依旧为

1. // 在T中找模板，text串的下标从0开始，长度为len
2. void find(char text[], int len) {
3. int j = 0; // 当前结点编号，初始为根结点
4. for(int i = 0; i < len; i++) { // 文本串当前指针
5. int c = idx(text[i]);
6. j = ch[j][c];
7. if(val[j]) print(j);
8. else if(last[j]) print(last[j]); // 找到了！
9. }
10. }

略加修改print函数为

1. // 递归打印以结点j结尾的所有字符串
2. void print(int j) { //输出j节点的信息，如果last[j]存在，last[j]的位置也有字符
3. if(j) {
4. cnt[j]++; //初始化都为0
5. print(last[j]);
6. }
7. }

这样，最后cnt[i]数组里，就表示以g[root,i]这个单词出现的次数。

1. Trie图的动态规划

这是一个坑爹的高频考点，先构建Trie图以后，大致问题都是要转化为类似图论DP之类的问题。要多考虑矩阵乘法。

1. 限制内存，需要左儿子右兄弟AC自动机

直接套用左儿子右兄弟版Trie，改为AC自动机即可。见附录

### 错误注意事项

1. 模板串是否会多次出现同样的？比如abc abc abc
2. 模板串为abc,abcde这种包含关系是否考虑？
3. 匹配串为aaaaaaa,，模板串为aaaa,aaa这种情况？

### 附录完整模板

#### 1、正常AC自动机：

1. const int SIGMA\_SIZE = 26;
2. const int MAXNODE = 10010;
3. /\*
4. \* AC自动机，令g[i,j]表示从i到j这一路遍历的所有字符串。 f[i]的意义就是g[?,i]和g[0,f[i]]的字符串是相等的
5. \* last[i] ,表示g[0,last[i]]的字符串，是确定存在的，并且以last[i]结尾的字符串\*/
6. struct AhoCorasickAutomata {
7. int ch[MAXNODE][SIGMA\_SIZE];
8. int f[MAXNODE]; // fail函数
9. int val[MAXNODE]; // 每个字符串的结尾结点都有一个非0的val
10. int last[MAXNODE]; // 输出链表的下一个结点
11. int cnt[1010][1010];
12. int sz;
13. queue<int>q;
14. int g[MAXNODE];//i节点，能匹配到哪个串
15. int pipei[1100];
16. void init() {
17. sz = 1;
18. memset(ch[0], 0, sizeof(ch[0]));
19. //memset(cnt, 0, sizeof(cnt));
20. memset(pipei,-1,sizeof(pipei));
21. }
22. // 字符c的编号
23. int idx(char c)
24. {
25. //if (c == '\0') return 62;
26. /\*
27. if (c >= '0' && c <= '9') return c - '0';
28. if (c >= 'a' && c <= 'z') return c - 'a' + 10;
29. return c - 'A' + 36;
30. \*/
31. return c-'a';
32. }
33. // 插入字符串。v必须非0
34. void insert(char s[], int len, int id) {
35. int now = 0;
36. for(int i = 0; i < len; i++) {
37. int c = idx(s[i]);
38. if(!ch[now][c]) {
39. memset(ch[sz], 0, sizeof(ch[sz]));
40. val[sz] = 0;
41. g[sz] = -1;
42. ch[now][c] = sz++;
43. }
44. now = ch[now][c];
45. }
46. val[now] =1;//这里有单词
47. pipei[id] = g[now];
48. g[now] = id;
49. }
50. // 在T中找模板，text串的下标从0开始，长度为len
51. void find(char text[], int len, int hang) {
52. int j = 0; // 当前结点编号，初始为根结点
53. for(int i = 0; i < len; i++) { // 文本串当前指针
54. int c = idx(text[i]);
55. j = ch[j][c];
56. if(val[j])
57. {
58. for (int tmp = g[j]; tmp != -1; tmp = pipei[tmp])
59. {
61. int lie = i;
62. if (hang - tmp < 0)continue;
63. if (lie-Y+1<0)continue;
64. cnt[hang-tmp][lie-Y+1]++;
65. }
66. }
67. }
68. }
69. //计算fail指针
70. void get\_fail()
71. {
72. f[0] = 0;//fail[i]表示，当匹配到某个位置失败，下一个自动的位置
73. for (int c = 0; c < SIGMA\_SIZE; c++)
74. {
75. int will = ch[0][c];
76. if (will)
77. {
78. f[will]=0;
79. q.push(will);
80. last[will] = 0;
81. }
82. }
83. while (!q.empty())
84. {
85. int now = q.front();
86. q.pop();
87. for (int c = 0; c < SIGMA\_SIZE; ++ c)
88. {
89. int will = ch[now][c]; //now节点，想要访问的下标
90. if (!will)
91. {
92. ch[now][c] = ch[f[now]][c];
93. continue;
94. }
95. q.push(will);
96. int pre = f[now]; //失配指针,先指now的失配，至少有一段都是相等的
97. while (pre && !ch[pre][c]) pre = f[pre];//往前跳失配指针，类似 KMP
98. f[will] = ch[pre][c]; // f[i]的意义就是g[?,i]和g[0,f[i]]的字符串是相等的
99. last[will] = val[f[will]] ? f[will] : last[f[will]];
100. }
101. }
102. }
103. void doit()
104. {
105. int ans = 0;
106. for (int i = 0; i < n; ++ i)
107. for (int j = 0; j < m; ++ j)
108. if (cnt[i][j] == X) ++ans;
109. printf("%d\n", ans);
110. }
111. }ac;

#### 2、左儿子右兄弟版AC自动机（节约内存，慢一倍）

1. const int MAXNODE = 1000010;
2. struct AhoCorasickAutomata {
3. int head[MAXNODE]; // head[i]为第i个结点的左儿子编号
4. int next[MAXNODE]; // next[i]为第i个结点的右兄弟编号
5. char ch[MAXNODE]; // ch[i]为第i个结点上的字符
6. int val[MAXNODE];
7. int last[MAXNODE];
8. int f[MAXNODE]; //fail指针
9. int sz; // 结点总数
10. int cnt[MAXNODE];
11. queue<int>q;
12. void init()
13. {
14. // 初始时只有一个根结点
15. sz = 1;
16. //tot[0] = 0;
17. head[0] = next[0] = 0;
18. memset(val, 0, sizeof(val));
19. memset(last, 0, sizeof(last));
20. memset(f, 0, sizeof(f));
21. memset(cnt,0,sizeof(cnt));
22. }
23. // 插入字符串s，沿途更新tot
24. void insert(char s[], int len) {
25. int now = 0, will;
26. //tot[0]++;
27. for(int i = 0; i < len; i++) {
28. // 找字符s[i]
29. bool found = false;
30. for(will = head[now]; will; will = next[will])
31. if(ch[will] == s[i]) // 找到了
32. {
33. found = true;
34. break;
35. }
36. if(!found)
37. {
38. will = sz++; // 新建结点
39. //tot[will] = 0;
40. ch[will] = s[i];
41. next[will] = head[now];
42. head[now] = will; // 插入到链表的首部
43. head[will] = 0;
44. }
45. now = will;
46. //tot[now]++;
47. }
48. val[now] ++ ;
49. }
50. int is\_son(int now, char t)//查找now节点，的t字符,有则返回下标，否则返回0(root)
51. {
52. //ch是否是now的儿子
53. for (int will = head[now]; will; will = next[will])
54. {
55. if (ch[will]==t) return will;
56. }
57. return 0;
58. }
60. void get\_fail()
61. {
62. f[0] = 0;//fail[i]表示，当匹配到某个位置失败，下一个自动的位置
63. for (int now = head[0]; now; now = next[now])
64. {
65. f[now]=0;
66. q.push(now);
67. last[now] = 0;
68. }
69. while (!q.empty())
70. {
71. int now = q.front();
72. q.pop();
73. for (int will = head[now]; will; will = next[will])
74. {
75. q.push(will);
76. int pre = f[now]; //失配指针,先指now的失配，至少有一段都是相等的
77. while (pre && !is\_son(pre,ch[will])) pre = f[pre];//往前跳失配指针，类似 KMP
78. f[will] = is\_son(pre, ch[will]); // f[i]的意义就是g[?,i]和g[0,f[i]]的字符串是相等的
79. last[will] = val[f[will]] ? f[will] : last[f[will]];
80. }
81. }
82. }
83. // 递归打印以结点j结尾的所有字符串
84. void print(int j) //输出j节点的信息，如果last[j]存在，last[j]的位置也有字符
85. {
86. if(j)
87. {
88. //vis[pos] = max(val[j], vis[pos]);
89. cnt[j]=1;
90. print(last[j]);
91. }
92. }
93. int idx(char ch)
94. {
95. if ('a' <= ch && ch <='z') return ch;
96. if ('A' <= ch && ch <='Z') return ch -'A'+'a';
97. return ch;
98. }
99. // 在T中找模板，text串的下标从0开始，长度为len
100. void find(char text[], int len) {
101. int j = 0; // 当前结点编号，初始为根结点
102. for(int i = 0; i < len; i++) { // 文本串当前指针
103. char c = text[i];
104. while (j && !is\_son(j, c)) j = f[j];
105. j = is\_son(j, c);
106. if(val[j]) print(j);
107. else if(last[j]) print(last[j]); // 找到了！
108. }
109. }
110. }ac;

## 快速傅里叶变换

### 一、FFT的功能

所谓FFT是用来求卷积的。假如给两个数组, (超出数组长度的元素自动补0)。 那么的结果为一个数组，假设,那么结果就是(下标从0开始)

其中，

的取值从0开始一直往后取，取到所需要的范围。

举个例子,上述两个数组, 的结果就是



显然到后面就都是0了，因为数组末尾也默认补0。

FFT的主要功能就是求这样2个数组相乘的卷积。

### 二、二维 FFT与二维卷积

显然就是求二维卷积。给出两个数组，形如





这两个数组的卷积结果为，的每一项为



其中，，。

举个例子：





那么求二维FFT的时候，要先把的维度变为一样的，新的数组的长，为原来两个数组的长之和，新数组的宽为原来两个数组的宽之和。矩阵多出的地方补0。如下图所示.





这样的二维FFT可以转化为一维来做的。假设原来数组的长（横着的）宽（竖着的）分别为与, 数组的长宽为与，那么新数组的长，宽。

方法1：使用一维FFT方法。

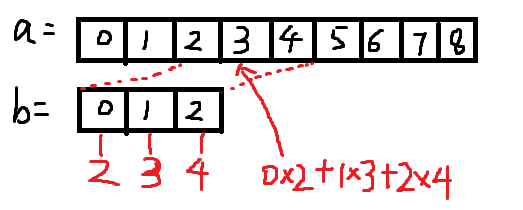
构造新数组与新数组。, 。然后直接做数组的FFT，假如原来，。那么

方法2：调用二维FFT

二维FFT则直接使用二维FFT的板子即可。

### 三、一维FFT与一维滤波器

1、举例说明什么叫滤波器



如图所示，是一个滤波器(长度为奇数，这样才有中心)，用在上“扫”一遍。上图中，已经扫到这个元素了，与的中间的对齐，然后每个对齐的元素对应相乘，然后再取一个和，那么得到了滤波后的值，我们通常保存在一个新数组里。对于对齐到的边界的情况，只需要根据题目要求（对齐不了的补0，或者补其他的）考虑即可。

假设为的“半径”，比如长度为3的时候，半径就是1。那么



2、一维滤波结果的求解

对于求数组的快速解法，依然是使用FFT。

(1) 数组长度都补为，的含义同上。

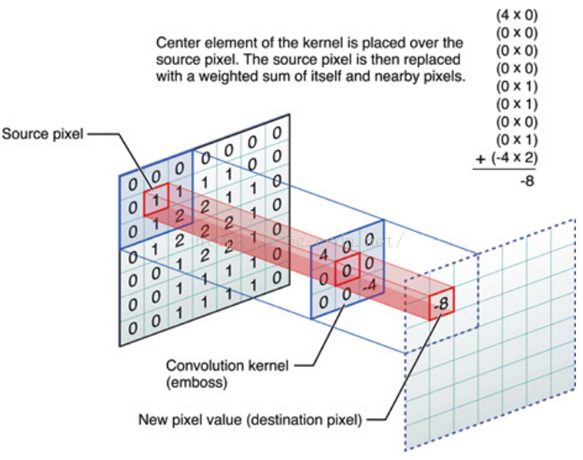
(2) 求FFT()，也就是，是的卷积。

(3) ,从0开始循环到即可获得所需数组。

### 四、二维滤波器

1、举例说明

如图所示,为一个二维滤波的例子。



二维滤波依旧需要一个有中心的滤波器。

2、二维滤波器的求解。

假设滤波器半径分为长半径与宽半径。比如(宽为5，长为3.竖着为宽，横着为长)，他的长半径就是1，宽半径为2。为原始二维数组，为二维滤波器。假设数组的长为,宽为，所有下标为从0开始。

求，用FFT即可

,按照i=0,i<。 的顺序即可。

### 附录

#### 一、代码

1. struct Complex
2. {
3. double x, y;
4. Complex(double x = 0., double y = 0.) : x(x), y(y) {}
5. Complex operator + (const Complex& rhs) const
6. {
7. return Complex(x + rhs.x, y + rhs.y);
8. }
9. Complex operator - (const Complex& rhs) const
10. {
11. return Complex(x - rhs.x, y - rhs.y);
12. }
13. Complex operator \* (const Complex& rhs) const
14. {
15. return Complex(x \* rhs.x - y \* rhs.y, x \* rhs.y + y \* rhs.x);
16. }
17. void operator /= (const double& rhs)
18. {
19. x /= rhs; y /= rhs;
20. }
21. };
22. void FFT(Complex P[], int n, int oper)
23. {
24. for (int i = 1, j = 0; i < n - 1; i++)
25. {
26. for (int s = n; j ^= s >>= 1, ~j & s;);
27. if (i < j) swap(P[i], P[j]);
28. }
29. Complex unit\_p0, unit;
30. for (int d = 0; (1 << d) < n; d++)
31. {
32. int m = 1 << d, m2 = m \* 2;
33. double p0 = PI / m \* oper;
34. unit\_p0 = Complex(cos(p0), sin(p0));
35. for (int i = 0; i < n; i += m2)
36. {
37. unit = 1;
38. for (int j = 0; j < m; j++)
39. {
40. Complex &P1 = P[i + j + m], &P2 = P[i + j];
41. Complex t = unit \* P1;
42. P1 = P2 - t;
43. P2 = P2 + t;
44. unit = unit \* unit\_p0;
45. }
46. }
47. }
48. if (oper == -1)
49. {
50. for (int i = 0; i < n; i++)
51. {
52. P[i] /= n;
53. }
54. }
55. }
56. //做一维FFT
57. //Complex类型的变量可以直接赋值double 结果保存在x中。
58. //例如 ttt = 5;//即ttt.x为5，ttt.y为0
59. //多项式a和多项式b相乘，结果保存在a中
60. //a,b的数组长度，需要为while (len < lenA + lenB) len <<= 1;的len长度
61. void multiply(Complex \*a, int lenA, Complex \*b, int lenB) //一维FFT。结果长度为lenA+lenB
62. {
63. int lenAns = 1;
64. while (lenAns < lenA + lenB) lenAns <<= 1;
65. FFT(a, lenAns, 1);
66. FFT(b, lenAns, 1);
67. for(int i = 0; i < lenAns; ++ i) a[i] = a[i] \* b[i];
68. FFT(a, lenAns, -1);
69. }
70. const int maxn = 1 << 10;
71. // while (maxn<=max(s+x,t+y)) maxn<<=1; 这就是maxn的取值
72. //这个函数只是为了FFT2D调用使用
73. Complex tmp[maxn];
74. void fft2D(Complex y[][maxn],int len,int on)
75. {
76. for(int i=0;i<len;i++) FFT(y[i],len,on);
77. for(int j=0;j<len;j++)
78. {
79. for(int i=0;i<len;i++) tmp[i]=y[i][j];
80. FFT(tmp,len,on);
81. for(int i=0;i<len;i++) y[i][j]=tmp[i];
82. }
83. }
84. //直接做二维FFT
85. //s t分别为a的行数和列数
86. //x y分别为b的行数和列数
87. //最后a数组会变为所需结果
88. Complex A[maxn][maxn], B[maxn][maxn];
89. void FFT2D(double a[][maxn], int s, int t, double b[][maxn], int x, int y)//二维FFT主函数
90. {
91. //得出的卷积行列分别为s+x-1与 t+y-1
92. //如果要做滤波器用的话，需要把b数组翻转180度
93. /\*
94. double c[maxn][maxn];//要定义在外面，不然炸栈空间
95. for (int i = 0; i < x; i++)
96. for (int j = 0; j < y; j++)
97. c[i][j] = b[i][j];
98. for (int i = 0; i < x; i++)
99. for (int j = 0; j < y; j++)
100. b[i][j] = c[x-i-1][y-j-1];
101. \*/
102. memset(A, 0, sizeof(A));
103. memset(B, 0, sizeof(B));
104. int len = 1;
105. while (len<=max(s+x,t+y)) len<<=1;
106. for (int i = 0; i < s; ++ i)
107. for (int j = 0; j <= t; ++ j)
108. A[i][j].x = a[i][j];
109. for (int i = 0; i < x; ++ i)
110. for (int j = 0; j < y; ++ j)
111. B[i][j].x = b[i][j];
112. fft2D(A, len, 1);
113. fft2D(B, len, 1);
114. for (int i = 0; i < len; ++ i)
115. for (int j = 0;j < len; ++ j)
116. A[i][j] = A[i][j] \* B[i][j];
117. fft2D(A,len,-1);
118. for (int i = 0; i < len; ++ i)
119. for (int j = 0; j < len ; ++ j)
120. a[i][j] = A[i][j].x;
121. }
122. //把二维转化成一维做的二维FFT  
     //用二维转一位时  
     //（A矩阵长+B矩阵长）\*（A矩阵宽\*B矩阵宽度）要小于maxn 且maxn为2的整数倍
123. //抄板子注意maxN与maxM的区别  
     Complex A[maxn], B[maxn];
124. int FFT2D(double a[][maxm], int s, int t, double b[][maxm], int x, int y)
125. {
126. memset(A, 0, sizeof(A));
127. memset(B, 0, sizeof(B));
128. int N = s + x, M = t + y;
129. for (int i = 0; i < s; ++ i)
130. for (int j = 0; j < t; ++ j)
131. A[i \* M + j] = a[i][j];
132. for (int i = 0; i < x; ++ i)
133. for (int j = 0; j < y; ++ j)
134. B[i \* M + j] = b[i][j];
135. multiply(A, N \* M, B, N \* M);
136. for (int i = 0; i < N; ++ i)
137. for (int j = 0; j < M; ++ j)
138. a[i][j] = A[i \* M + j].x;
139. }
140. //直接调用FFT函数，和FFT用法一样
141. struct NTT {
142. const static LL mod = (1LL << 47) \* 7 \* 4451 + 1, g = 3;//数字不用改
143. //4384957924686954497(大约1<<62
144. LL mul(LL x , LL y) {
145. return (x \* y - (LL)(x / (long double)mod \* y + 1e-3) \* mod + mod) % mod;
146. }
147. LL powMod(LL a, LL b) {
148. LL res = 1, tmp = a;
149. while (b) {
150. if (b & 1) res = mul(res, tmp);
151. tmp = mul(tmp, tmp);
152. b >>= 1;
153. }
154. return res;
155. }
156. void DFT(LL y[], int n, bool rev) {
157. for (int i = 1, j, t, k; i < n; i++) {
158. for (k = n >> 1, t = i, j = 0; k; k >>= 1, t >>= 1) {
159. j = j << 1 | t & 1;
160. }
161. if (i < j) swap(y[i], y[j]);
162. }
163. for (int s = 2, ds = 1; s <= n; ds = s, s <<= 1) {
164. LL wn = powMod(g, (mod - 1) / s);
165. if (!rev) wn = powMod(wn, mod - 2);
166. for (int k = 0; k < n; k += s) {
167. LL w = 1, t;
168. for (int i = k; i < k + ds; ++ i, w = mul(w, wn)) {
169. y[i + ds] = (y[i] - (t = mul(y[i + ds], w)) + mod) % mod;
170. y[i] = (y[i] + t) % mod;
171. }
172. }
173. }
174. }
175. void FFT(LL x1[], int lenA, LL x2[], int lenB) {
176. int lenAns;
177. for (lenAns = 1; lenAns < lenA + lenB; lenAns <<= 1);
178. DFT(x1, lenAns, 1);
179. DFT(x2, lenAns, 1);
180. for (int i = 0; i < lenAns; i++) x1[i] = mul(x1[i], x2[i]);
181. DFT(x1, lenAns, 0);
182. LL vn = powMod(lenAns, mod - 2);
183. for (int i = 0; i < lenAns; i++) x1[i] = mul(x1[i], vn);
184. }
185. } g;

#### FFT 可能用到的各种素数

FFT，由于是在模意义下的，需要各种素数……

然后就打了个表方便以后查了、

如果  是个素数，那么在意义下，可以处理  以内规模的数据，

是一个挺好的数，平方刚好不会爆 long long

加起来刚好不会爆 int 也不错

下面是刚刚打出来的表格（g 是的原根）

|  |  |  |  |
| --- | --- | --- | --- |
|  | r | k | g |
| 3 | 1 | 1 | 2 |
| 5 | 1 | 2 | 2 |
| 17 | 1 | 4 | 3 |
| 97 | 3 | 5 | 5 |
| 193 | 3 | 6 | 5 |
| 257 | 1 | 8 | 3 |
| 7681 | 15 | 9 | 17 |
| 12289 | 3 | 12 | 11 |
| 40961 | 5 | 13 | 3 |
| 65537 | 1 | 16 | 3 |
| 786433 | 3 | 18 | 10 |
| 5767169 | 11 | 19 | 3 |
| 7340033 | 7 | 20 | 3 |
| 23068673 | 11 | 21 | 3 |
| 104857601 | 25 | 22 | 3 |
| 167772161 | 5 | 25 | 3 |
| 469762049 | 7 | 26 | 3 |
| 1004535809 | 479 | 21 | 3 |
| 2013265921 | 15 | 27 | 31 |
| 2281701377 | 17 | 27 | 3 |
| 3221225473 | 3 | 30 | 5 |
| 75161927681 | 35 | 31 | 3 |
| 77309411329 | 9 | 33 | 7 |
| 206158430209 | 3 | 36 | 22 |
| 2061584302081 | 15 | 37 | 7 |
| 2748779069441 | 5 | 39 | 3 |
| 6597069766657 | 3 | 41 | 5 |
| 39582418599937 | 9 | 42 | 5 |
| 79164837199873 | 9 | 43 | 5 |
| 263882790666241 | 15 | 44 | 7 |
| 1231453023109121 | 35 | 45 | 3 |
| 1337006139375617 | 19 | 46 | 3 |
| 3799912185593857 | 27 | 47 | 5 |
| 4222124650659841 | 15 | 48 | 19 |
| 7881299347898369 | 7 | 50 | 6 |
| 31525197391593473 | 7 | 52 | 3 |
| 180143985094819841 | 5 | 55 | 6 |
| 1945555039024054273 | 27 | 56 | 5 |
| 4179340454199820289 | 29 | 57 | 3 |

## JAVA

### JAVA程序大体框架

1. IO部分以及一些小框架

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.PrintWriter;

import java.util.StringTokenizer;

public class Main {

static String input;

static InputReader reader = new InputReader();

static PrintWriter writer = new PrintWriter(System.out);

public static void main(String args[]) throws IOException

{

input = reader.nextString();

writer.print(input);

if (doit())

System.out.print("YES\n");

else System.out.print("NO\n");

writer.close();//程序结束一定要这句话

}

public static boolean doit()

{

int len = input.length();

String aim = new String("CODEFORCES");

String tmp = new String("");

if (len >= 5)

{

//??

}

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

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

{

tmp = input.substring(0, i);

tmp = tmp.concat(input.substring(j));

if (tmp.equals(aim)) return true;

}

return false;

}

}

class InputReader

{

public InputReader() {

// TODO Auto-generated constructor stub

tokenizer = new StringTokenizer("");

reader = new BufferedReader(new InputStreamReader(System.in));

}

public String nextTokenizer() throws IOException

{

while (!tokenizer.hasMoreTokens())

{

tokenizer = new StringTokenizer(reader.readLine());

}

return tokenizer.nextToken();

}

public int nextInt() throws IOException

{

return Integer.valueOf(nextTokenizer());

}

public String nextString() throws IOException

{

return nextTokenizer();

}

private StringTokenizer tokenizer;

private BufferedReader reader;

}

### 简单的大数例子

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.PrintWriter;

import java.math.BigDecimal;

import java.math.BigInteger;

import java.util.StringTokenizer;

public class Main {

static String *input*;

static InputReader *reader* = new InputReader();

static PrintWriter *writer* = new PrintWriter(System.*out*);

public static void main(String args[]) throws IOException

{

//input = reader.nextString();

BigInteger x = new BigInteger("123");

BigInteger y = new BigInteger("5");

*writer*.println(x.multiply(y));//乘法

*writer*.println(x.divide(y));//除非

*writer*.println(x.add(y));//加法

*writer*.println(x.modPow(new BigInteger("100000000000"), new BigInteger("999983")));//这个不慢

String q = new String((new Integer(5)).toString());

*writer*.println(q);

*writer*.println(*pi*);

*writer*.close();//程序结束一定要这句话

}

static BigDecimal *pi* = new BigDecimal(Math.*acos*(-1));

}

### 二、BigDecimal大小数

大数用来卡过一些精度题，会比较爽。

1. 四舍五入N位小数

static BigDecimal *pi* = new BigDecimal(Math.*acos*(-1));

//四舍五入2位小数，转为double

double pk = *pi*.setScale(2, BigDecimal.*ROUND\_HALF\_UP*).doubleValue();

1. 精度控制

下列的写法会出问题，因为出现了无限循环小数1/3

BigDecimal a1 = new BigDecimal(1);

BigDecimal a3 = new BigDecimal(3);

a1.divide(a3);

*writer*.println(a1);

解决的方法，就是要进行精度控制。

BigDecimal a1 = new BigDecimal(1);

BigDecimal a3 = new BigDecimal(3);

a1 = a1.divide(a3, 200, BigDecimal.*ROUND\_CEILING*);

*writer*.println(a1+" "+a3);

精度控制他们运算的结果只有200位有效数字，以及精度最终向有效位进位方式。

### 三、JAVA的一些STL小技巧

1. map的使用

比赛经常用到HashMap与TreeMap。

先介绍TreeMap，二叉树Map嘛，自然就是log级各种运算的时间。所以对自定义结构体需要定义比较函数。

/\*常用的包含x,y节点的node的class\*/

class nodexy

{

int x,y;

nodexy(){}

nodexy(int xx, int yy)

{

x = xx;

y = yy;

}

}

/\*使用这样的比较函数\*/

static Comparator<nodexy> cmp = new Comparator<nodexy>()

{

public int compare(nodexy o1, nodexy o2) {

// TODO Auto-generated method stub

if (o1.x==o2.x) return o1.y-o2.y;

return o1.x-o2.x;

}

};

下面是一个简单的TreeMap的样例

public static void main(String args[]) throws IOException

{

Map<Integer, Integer>q = new TreeMap<Integer, Integer>(); //int,int类型的TreeMap,没有包含比较器！

q.put(1, 1);

q.put(2, 2);

q.put(3, 3);

q.put(3, 4); //重复的key的value只保留最后一个~ key才是关键字

q.put(-1,0);

q.put(-11,4);

q.put(12,2);

@SuppressWarnings("rawtypes") //消除下面的警告信息

Iterator it = q.entrySet().iterator(); //和C++一样用迭代器遍历

System.out.print("size = " + q.size()+"\n"); //总元素数量

while (it.hasNext())

{

@SuppressWarnings("unchecked") //消除下面的警告信息

Map.Entry<Integer, Integer> ent = (Entry<Integer, Integer>) it.next();

int key = ent.getKey(); //key

int value = ent.getValue(); //value

System.out.println(key+" " + value);

}

}

或者是这样的例子

public static void main(String args[]) throws IOException

{

TreeMap<Integer, Integer>q = new TreeMap<Integer, Integer>(); //int,int类型的TreeMap,没有包含比较器！

q.put(1, 1);

q.put(2, 2);

q.put(3, 3);

q.put(3, 4); //重复的key的value只保留最后一个~ key才是关键字

q.put(-1,0);

q.put(-11,4);

q.put(12,2);

@SuppressWarnings({ "rawtypes", "unused" }) //消除下面的警告信息

Iterator it = q.entrySet().iterator(); //和C++一样用迭代器遍历

System.out.print("size = " + q.size()+"\n"); //总元素数量

/\*

\* 不需要输出一次啦,直接这样

while (it.hasNext())

{

@SuppressWarnings("unchecked") //消除下面的警告信息

Map.Entry<Integer, Integer> ent = (Entry<Integer, Integer>) it.next();

int key = ent.getKey(); //key

int value = ent.getValue(); //value

System.out.println(key+" " + value);

}

\*/

int ans = q.get(12);

System.out.println(ans); //输出了2

/\*对于不存在的键值的处理， 先判断是否是null\*/

if (q.get(13) != null)

{

ans = q.get(13);

System.out.println(ans);

}else System.out.println("nothing");//输出了nothing

System.out.println(q.higherKey(4));//比4大的是12，所以输出12

System.out.println(q.higherKey(12));//比12大的没有，输出null

System.out.println(q.lastKey()); //输出最后一个key,也就是12，也是对的

q.remove(2); //删除键为2的

if (q.get(2) != null)

{

ans = q.get(2);

System.out.println(ans);

}else System.out.println("nothing");//因为2被删了，所以输出了nothing

}

关于遍历，访问，删除迭代器指向内容的话，例子如下所示。

public static void main(String args[]) throws IOException

{

TreeMap<Integer, Integer>q = new TreeMap<Integer, Integer>(); //int,int类型的TreeMap,没有包含比较器！

q.put(1, 1);

q.put(2, 2);

q.put(3, 3);

q.put(3, 4); //重复的key的value只保留最后一个~ key才是关键字

q.put(-1,0);

q.put(-11,4);

q.put(12,2);

@SuppressWarnings({ "rawtypes", "unused" }) //消除下面的警告信息

Iterator it = q.entrySet().iterator(); //和C++一样用迭代器遍历

System.out.print("size = " + q.size()+"\n"); //总元素数量

while (it.hasNext())

{

@SuppressWarnings("unchecked") //消除下面的警告信息

Map.Entry<Integer, Integer> ent = (Entry<Integer, Integer>) it.next();

int key = ent.getKey(); //key

int value = ent.getValue(); //value

if (key==-1 || key == 1 || key==12)it.remove(); //删除迭代器这个值，并且向后迭代！

System.out.println(key+" " + value);

}

System.out.println();

System.out.println();

/\*再出输出一次，发现-1，1,12都已经被删啦！\*/

it = q.entrySet().iterator();

while (it.hasNext())

{

@SuppressWarnings("unchecked") //消除下面的警告信息

Map.Entry<Integer, Integer> ent = (Entry<Integer, Integer>) it.next();

int key = ent.getKey(); //key

int value = ent.getValue(); //value

System.out.println(key+" " + value);

}

}

/\*下面例子依然是TreeMap,但是我们修改了key为一个class, 同时设置了比较函数~，比较经典\*/

class Pack

{

int a, b;

Pack(){}

Pack(int x, int y)

{

a=x;

b=y;

}

boolean same(Pack tmp)

{

if (tmp.a == a && tmp.b==b) return true;

return false;

}

};

public class Doit {

//一边访问，一边删除迭代器指向的内容

static Comparator<Pack> *cmp* = new Comparator<Pack>() {

@Override

public int compare(Pack o1, Pack o2) {

// TODO Auto-generated method stub

if (o1.a==o2.a) return o1.b-o2.b;

return o1.a-o2.a;

}

};

public static void main(String args[]) throws IOException

{

TreeMap<Pack, Integer>q = new TreeMap<Pack, Integer>(*cmp*); //int,int类型的TreeMap,没有包含比较器！

q.put(new Pack(1,1), 1);

q.put(new Pack(2,2), 2);

q.put(new Pack(2,3), 3);

q.put(new Pack(1,-1), 4); //重复的key的value只保留最后一个~ key才是关键字

q.put(new Pack(5,2),0);

q.put(new Pack(5,3),4);

q.put(new Pack(4,2),2);

q.put(new Pack(4,2),2);

@SuppressWarnings("rawtypes")

Iterator it = q.entrySet().iterator(); //和C++一样用迭代器遍历

System.*out*.print("size = " + q.size()+"\n"); //总元素数量，输出为7个

while (it.hasNext())

{

@SuppressWarnings("unchecked") //消除下面的警告信息

Map.Entry<Pack, Integer> ent = (Entry<Pack, Integer>) it.next();

Pack key = ent.getKey();

int key1 = ent.getKey().a; //key

int key2 = ent.getKey().b;

int value = ent.getValue(); //value

if (key.same(new Pack(5,3)) || key.same(new Pack(2,2)))it.remove(); //删除迭代器这个值，并且向后迭代！

System.*out*.println(key1+" "+key2+" " + value);

}

System.*out*.println();

System.*out*.println();

/\*再出输出一次，发现-1，1,12都已经被删啦！\*/

it = q.entrySet().iterator();

while (it.hasNext())

{

@SuppressWarnings("unchecked") //消除下面的警告信息

Map.Entry<Pack, Integer> ent = (Entry<Pack, Integer>) it.next();

Pack key = ent.getKey();

int key1 = ent.getKey().a; //key

int key2 = ent.getKey().b;

int value = ent.getValue(); //value

System.*out*.println(key1+" "+key2+" " + value);

}

}

}

1. 实现c++的vector功能的arraylist

讲道理，arraylist还是蛮重要的一个东西。直接举例子说明即可

class Pack

{

int a, b;

Pack(){}

Pack(int x, int y)

{

a=x;

b=y;

}

boolean same(Pack tmp)

{

if (tmp.a == a && tmp.b==b) return true;

return false;

}

};

public class Doit {

//一边访问，一边删除迭代器指向的内容

static Comparator<Pack> *cmp* = new Comparator<Pack>() {

@Override

public int compare(Pack o1, Pack o2) {

// TODO Auto-generated method stub

if (o1.a==o2.a) return o1.b-o2.b;

return o1.a-o2.a;

}

};

public static void main(String args[]) throws IOException

{

ArrayList<Pack>q = new ArrayList<Pack>();

q.add(new Pack(5,5));//塞入队尾

q.add(new Pack(4,4));

q.add(new Pack(7,7));

System.*out*.println(q.get(0).a + " " + q.get(0).b);

System.*out*.println(q.get(1).a + " " + q.get(1).b);

System.*out*.println(q.get(2).a + " " + q.get(2).b);

//Pack sb[] = q.toArray();

Pack sb[];

sb = q.toArray(new Pack[q.size()]); //转换成数组

for (int i = 0; i != q.size(); ++ i)

{

System.*out*.println(sb[i].a);

}

}

}

1. 关于数组排序

数组排序也是很常见的事情。

class Pack

{

int a, b;

Pack(){}

Pack(int x, int y)

{

a=x;

b=y;

}

boolean same(Pack tmp)

{

if (tmp.a == a && tmp.b==b) return true;

return false;

}

};

public class Doit {

//一边访问，一边删除迭代器指向的内容

static Comparator<Pack> *cmp* = new Comparator<Pack>() {

@Override

public int compare(Pack o1, Pack o2) {

// TODO Auto-generated method stub

if (o1.a==o2.a) return o1.b-o2.b;

return o1.a-o2.a;

}

};

public static void main(String args[]) throws IOException

{

ArrayList<Pack>q = new ArrayList<Pack>();

q.add(new Pack(5,5));//塞入队尾

q.add(new Pack(4,4));

q.add(new Pack(7,7));

q.add(new Pack(4,3));

q.add(new Pack(4,2));

q.add(new Pack(4,10));

q.add(new Pack(4,4));

/\*访问arraylist内元素方法用get\*/

System.*out*.println(q.get(0).a + " " + q.get(0).b);

System.*out*.println(q.get(1).a + " " + q.get(1).b);

System.*out*.println(q.get(2).a + " " + q.get(2).b);

//Pack sb[] = q.toArray();

/\*arraylist转化为数组\*/

Pack sb[];

sb = q.toArray(new Pack[q.size()]); //转换成数组

/\*

for (int i = 0; i != q.size(); ++ i)

{

System.out.println(sb[i].a);

}

\*/

/\*数组排序的方法\*/

Arrays.sort(sb,0, q.size(),*cmp*); //排序，[L,R)区间进行排序。 例子是[0,q.size()) 之间的数字进行排序

for (int i = 0; i != q.size(); ++ i) //输出了数组中的元素，从小到大

{

System.*out*.println(sb[i].a + " " + sb[i].b);

}

}

}

1. 容器的排序

class Pack

{

int a, b;

Pack(){}

Pack(int x, int y)

{

a=x;

b=y;

}

boolean same(Pack tmp)

{

if (tmp.a == a && tmp.b==b) return true;

return false;

}

};

public class Doit {

//一边访问，一边删除迭代器指向的内容

static Comparator<Pack> *cmp* = new Comparator<Pack>() {

@Override

public int compare(Pack o1, Pack o2) {

// TODO Auto-generated method stub

if (o1.a==o2.a) return o1.b-o2.b;

return o1.a-o2.a;

}

};

public static void main(String args[]) throws IOException

{

ArrayList<Pack>q = new ArrayList<Pack>();

q.add(new Pack(5,5));//塞入队尾

q.add(new Pack(4,4));

q.add(new Pack(7,7));

q.add(new Pack(4,3));

q.add(new Pack(4,2));

q.add(new Pack(4,10));

q.add(new Pack(4,4));

q.add(new Pack(2,4));

/\*访问arraylist内元素方法用get\*/

System.*out*.println(q.get(0).a + " " + q.get(0).b);

System.*out*.println(q.get(1).a + " " + q.get(1).b);

System.*out*.println(q.get(2).a + " " + q.get(2).b);

//Pack sb[] = q.toArray();

System.*out*.println();

System.*out*.println();

System.*out*.println();

Collections.sort(q,*cmp*);

Iterator it = q.iterator();

while (it.hasNext())

{

Pack tmp = (Pack) it.next();

System.*out*.println(tmp.a+" "+tmp.b);

}

}

}

另一种做比较器的方法来排序，用接口实现

class Pack

{

int a, b;

Pack(){}

Pack(int x, int y)

{

a=x;

b=y;

}

boolean same(Pack tmp)

{

if (tmp.a == a && tmp.b==b) return true;

return false;

}

};

class MyCmp implements Comparator<Pack>

{

//@Override

public int compare(Pack o1, Pack o2) {

// TODO Auto-generated method stub

if (o1.a==o2.a) return o1.b-o2.b;

return o1.a - o2.a;

}

}

public class Doit {

//一边访问，一边删除迭代器指向的内容

static Comparator<Pack> *cmp* = new MyCmp();

public static void main(String args[]) throws IOException

{

ArrayList<Pack>q = new ArrayList<Pack>();

q.add(new Pack(5,5));//塞入队尾

q.add(new Pack(4,4));

q.add(new Pack(7,7));

q.add(new Pack(4,3));

q.add(new Pack(4,2));

q.add(new Pack(4,10));

q.add(new Pack(4,4));

q.add(new Pack(2,4));

/\*访问arraylist内元素方法用get\*/

System.*out*.println(q.get(0).a + " " + q.get(0).b);

System.*out*.println(q.get(1).a + " " + q.get(1).b);

System.*out*.println(q.get(2).a + " " + q.get(2).b);

//Pack sb[] = q.toArray();

System.*out*.println();

System.*out*.println();

System.*out*.println();

Collections.sort(q,*cmp*);

Iterator it = q.iterator();

while (it.hasNext())

{

Pack tmp = (Pack) it.next();

System.*out*.println(tmp.a+" "+tmp.b);

}

}

}

1. queue使用方法

方法大致如下所示

static Queue<Coordinate> *queue* = new LinkedList<Coordinate>();//队列必须new LinkedList类型

queue.offer(new Coordinate(arg0, arg1)); 塞入新元素

queue.poll();//弹出头元素

long nowx = queue.peek().x; //peek读取头元素

POJ 2386的一个BFS题目的例子如下

public class Main

{

static InputReader reader = new InputReader(System.in);

static PrintWriter writer = new PrintWriter(System.out);

static long n, m;

static char map[][] = new char [150][150];

static Queue<Coordinate> queue = new LinkedList<Coordinate>();

static boolean vis[][] = new boolean[150][150];

final static long dx[] = {0, 0, 1, -1, 1, 1, -1, -1};

final static long dy[] = {-1, 1, 0, 0, -1, 1, -1, 1};

//false为没访问过，true为访问过

public static void main(String args[])

{

try

{

init();

doit();

}catch (IOException e){

System.out.println("chuwenti");

}

finally

{

writer.close();

}

}

static void init() throws IOException

{

for (int i = 0; i != 150; ++ i)

for (int j = 0; j != 150; ++ j)

vis[i][j] = false;

n = reader.nextInt();

m = reader.nextInt();

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

{

String tmp = reader.nextString();

map[i] = tmp.toCharArray();

}

}

static void bfs(long arg0, long arg1)

{

queue.offer(new Coordinate(arg0, arg1));

while (!queue.isEmpty())

{

long nowx = queue.peek().x;

long nowy = queue.peek().y;

queue.poll();

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

{

long willx = nowx + dx[i];

long willy = nowy + dy[i];

if (willx <0 || willy <0 || willx >= n || willy >= m) continue;

if (map[(int) willx][(int) willy] == '.') continue;

queue.offer(new Coordinate(willx, willy));

map[(int) willx][(int) willy] = '.';

}

}

}

static void doit()

{

long ans = 0;

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

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

{

if (map[i][j] == '.') continue;

bfs(i, j);

ans ++;

}

writer.println(ans);

}

}

1. 注意事项

(1)Heap的构造函数小注意点

static Comparator<Integer> *myCmp* = new MyCmp();

static PriorityQueue<Integer> q = new PriorityQueue<Integer>(mkh,myCmp);

注意，构造函数第一个需要填写数字。否则在部分OJ编译失败。并且这个数字大小写11即可。也就是mkh写上11。

(2) 关于new 对象数组

static Animal animal[] = new Animal[50010];

for (int i = 0; i != 50010; ++ i) animal[i] = new Animal();

必须得用上述方法进行初始化，否则实际上并没有初始化。

## KDTree

### 简介

所谓KDTree的主要功能是，给出一些D元组。比如，如果题目是若干个3元组的话，那么就是<1,4,8>,<14,44,66>,<9,0,-5>,<5,9,1>这样的一些组。如果看为三维坐标的话，那么每个点之间会有一个欧几里得距离，

KDTree解决的，就是任意给出一个点p<x,y,z……>，寻找一个距离这个点最近、第二近、第三近……的那个点。只能求最近，不能求远，因为回溯估价会出问题。

插入操作非常花费内存！只能插入大约5万这样数量级的数据。否则超出内存限制。并且内存大致使用量不方便计算。

对于插入操作：

插入1e4个节点，大概会申请5e4个Point

插入1e5个节点，大概会申请8e5个Point

插入1e6个节点，大概会申请12e6个Point

……

所以大致只能做到5e4这样的数量级的插入操作。当然不使用插入操作的话，内存空间就是O(n)而已。

### KDTree的构建

代码中有2个方法。

首先来简单解读代码。

1. const int D = 5;// 定义了每个点，最多为D元组。
2. struct Point { int x[D]; } buf[N];// 定义了buf数组
3. //可以先给buf数组赋值（赋上各种D元组） ，然后
4. KDTree kdt; //定义一颗kdtree
5. //记得，要给d赋值哦！d是实际读入的每一个坐标的维度
6. for (int i = 0; i < n; ++ i)
7. for (int j = 0; j < d; ++ j)
8. scanf("%d", &buf[i].x[j]);//读入n个点对，并且赋值进buf数组
9. initNull();//初始化kdtree
10. kdt = KDTree(buf, n);//给kdtree初始状态进行赋值。

上述是方法1，整个板子支持“动态插入”。所以，也就是可以进行插入操作。

举个例子

1. KDTree kdt; //定义一颗kdtree
2. for (int i = 0; i < n; ++ i)
3. for (int j = 0; j < d; ++ j)
4. scanf("%d", &buf[i].x[j]);//读入n个点对，并且赋值进buf数组
5. initNull();//初始化kdtree
6. kdt = KDTree(buf, n/4);
7. for (int i = n/4; i < n; ++ i)
8. kdt.insert(Point({buf[i]}));

这段代码和上述不同的地方在于，先强行构建一小部分，剩下的利用kdtree的insert函数，进行动态插入。

### KDTree更新信息。

删除操作，我们搜到的板子并不支持……但是可以利用一个set或者hash等方法，来标记一下这个节点已经被删掉了。

板子里有一个void updateAns(Point u, Point p)函数，。u是kdtree里的节点，P是query所询问的坐标。在这里，我们会根据需求，遍历到很多u节点。KDTree维护了大量节点，但是在检索的时候，还是有点暴力搜索的意思。所以会遍历到很多可能是“最近”或者“最远”点的u节点。我们把这些节点信息保存下来，放入一个堆里（求K近）。或者只更新最优解（求最近）。

1. void updateAns(Point u, Point p) {
2. // 在这里更新答案
3. // u是kdtree的节点，P是询问的坐标
4. // 求第ask\_m近
5. long long dis = u-p;
6. if (HEAP.size() < ask\_m) //堆是否有ask\_m个元素
7. {
8. HEAP.push(make\_pair(dis, u));//如果没有，那么进入新元素
9. }else //如果已经有ask\_m个元素了，那么距离P最远的节点出堆，加入新的元素
10. {
11. if (HEAP.top().first > dis)
12. {
13. HEAP.pop();
14. HEAP.push(make\_pair(dis, u));
15. ret = HEAP.top().first;//ret为距离P第ask\_m近的元素的距离
16. }
17. }
18. }

上述代码有一个ret的变量，这个变量可以作为后面控制回溯的剪枝。

### 询问操作

代码如下：

1. void query(Node \*t, Point p) {
2. if (t == null) return;
3. updateAns(t->val, p); //遍历到一个节点，去update一下堆里的元素
4. long long evalLeft = calcEval(p, t->ch[0], t->depth);//估算该节点左儿子的距离
5. long long evalRight = calcEval(p, t->ch[1], t->depth);//估算该节点右儿子的距离
6. if (evalLeft <= evalRight) {//简单的确定回溯先后的顺序,先遍历小的,贪心而已
7. //估价已经偏小了，如果ret依旧比估价小，说明答案不在这个分支里
8. //所以这里必须满足ret>估价，才能进行回溯搜索
9. if (ret > evalLeft) query(t->ch[0], p);
10. if (ret > evalRight) query(t->ch[1], p);
11. } else {
12. if (ret > evalRight) query(t->ch[1], p);
13. if (ret > evalLeft) query(t->ch[0], p);
14. }
15. }
16. void query(Point p) {
17. query(root, p); //去计算求解
18. //下面的代码，就是输出堆中的元素而已。
19. Point pt[20];
20. for (int j = 0; !HEAP.empty(); ++ j)
21. {
22. pt[j] = HEAP.top().second;
23. HEAP.pop();
24. }
25. for (int j = ask\_m-1;j>=0;--j) print(pt[j]);
26. }

询问操作主要就是问的时候会调用update的函数，来更新答案。

### 插入与删除

插入操作已经介绍过了，直接简单的insert一下即可。删除操作，则需要用map或者hash之类的方法来记录一下被删除的坐标，从而进行删除操作。如果重新被添加的话，一定要再map和hash里修改！

Hiti：涉及删除操作要用快速IO来抢时间！

### 注意事项

1. ret再query之前要清无穷大！或者根据题目修改！
2. 使用时注意先 initNull
3. 插入的空间复杂度 nlogn！（虽然实际没这么大，但是也很大了）
4. 不需要插入的时候可以简化代码。尽量不要插入！很慢的！
5. calcEval函数，的返回值，要根据题目修改！有的题目不是欧几里得是曼哈顿距离！
6. kdt = KDTree(buf, n);这个步骤，会修改buf数组里排列的顺序！

### 例题介绍

1. HDU 4347

题目大意：

多组数据，先给一些D元组。然后读入T组问题。每个问题分别是，先给一个D元组P，然后给一个ask\_m,问你P距离最开始给的一大堆D元组里最近的ask\_m个元组分别是什么。按照距离近远的顺序输出。

样例：

3 2

1 1

1 3

3 4

2

2 3

*2*

2 3

*1*

样例解释：

第1行：有3个2元组

第2~5行：每行都是一个2元组，一共3行。

第6行：有2个query

第7,8行：给一个2元组，<2,3>。输出距离<2,3>最近的2个2元组。

第9,10行：给一个2元组，<2,3>，输出距离<2,3>最近的1个2元组

代码：

1. #include <cstdio>
2. #include <queue>
3. #include <vector>
4. #include <iostream>
5. #include <algorithm>
6. using namespace std;
7. // 带插入版本 , 没有写内存回收 , 空间复杂度 nlogn , 如果不需要插入可以大大简化
8. // N 为最大点数, D 为每个点的最大维度, d 为实际维度
9. // 以查找最近点为例 ret 为当前最近点的距离的平方 , 用来剪枝 , 查询 k 近或 k 远的方法类似
10. // 使用时注意先 initNull
11. const long long INF = (int)1e9 + 10;
12. const int N = 2000000 + 10;
13. const int D = 5;
14. const double SCALE = 0.75;
15. struct Point { int x[D]; } buf[N];
16. int d;
17. long long ret;
18. int ctr;
19. long long operator - (const Point &A, const Point &B)//计算A,B两点距离的平方！平方
20. {
21. long long ret=0;
22. for (int i = 0; i < d; ++ i)
23. ret += (1LL\*A.x[i]-B.x[i]) \* (1LL\*A.x[i]-B.x[i]);
24. return ret;
25. }
26. bool operator < (const Point &a, const Point &b)
27. {
28. return a.x[ctr] < b.x[ctr];
29. }
30. struct Node {
31. int depth, size;
32. Node \*ch[2], \*p;
33. Point val, maxv, minv;
34. void set(Node \*t, int d) { ch[d] = t; t->p = this; }// set(t,1)表示让当前节点的1儿子为t。 就是修改当前节点的儿子指针用的
35. bool dir() { return this == p->ch[1]; } //和insert有关系
36. bool balanced() {//判断是否平衡 用途暂时不明
37. return (double)max(ch[0]->size, ch[1]->size) <= (double)size \* SCALE;
38. }
39. void update() {
40. size = ch[0]->size + ch[1]->size + 1; //更新当前节点节点总数
41. for(int i = 0; i < d; ++ i) {
42. maxv.x[i] = max(val.x[i], max(ch[0]->maxv.x[i], ch[1]->maxv.x[i]));//val是自己的值，maxv是所有左右儿子最大的val的值
43. minv.x[i] = min(val.x[i], min(ch[0]->minv.x[i], ch[1]->minv.x[i]));
44. }
45. }
46. } nodePool[N], \*totNode, \*null;
47. Node\* newNode(Point p, int depth) {//从内存池里拿出节点，这个节点用depth维
48. Node \*t = totNode ++;
49. t->ch[0] = t->ch[1] = t->p = null;
50. t->depth = depth;
51. t->val = t->maxv = t->minv = p;
52. t->size = 1;
53. return t;
54. }
55. void print(const Point &a)
56. {
57. for (int i = 0; i < d-1; ++ i)
58. printf("%d ", a.x[i]);
59. printf("%d\n", a.x[d-1]);
60. }
61. //int cmp(const Point &a, const Point &b) { return a.x[ctr] < b.x[ctr]; }
62. typedef pair<double,Point> dP;
63. priority\_queue<dP>HEAP;
64. int ask\_m;
65. struct KDTree {
66. Node \*root;
67. KDTree() { root = null; }
68. KDTree(Point \*a, int n) {
69. root = build(a, 0, n - 1, 0);
70. }
71. Node \*build(Point \*a, int l, int r, int depth) {
72. if (l > r) return null;
73. ctr = depth; //ctr是全局变量，可以控制到底是第几维
74. sort(a + l, a + r + 1);
75. int mid = (l + r) >> 1;
76. Node \*t = newNode(a[mid], depth);
77. t->set(build(a, l, mid - 1, (depth + 1) % d), 0);
78. t->set(build(a, mid + 1, r, (depth + 1) % d), 1);
79. t->update();
80. return t;
81. }
82. void tranverse(Node \*t, Point \*vec, int &tot) {
83. if (t == null) return;
84. vec[tot ++] = t->val;
85. tranverse(t->ch[0], vec, tot);
86. tranverse(t->ch[1], vec, tot);
87. }
88. void rebuild(Node \*t) {
89. Node \*p = t->p;
90. int tot = 0;
91. tranverse(t, buf, tot);
92. Node \*u = build(buf, 0, tot - 1, t->depth);
93. p->set(u, t->dir());
94. for( ; p != null; p = p->p) p->update();
95. if (t == root) root = u;
96. }
97. void insert(Point p) {
98. if (root == null) { root = newNode(p, 0); return; }
99. Node \*cur = root, \*last = null;
100. int dir = 0;
101. for( ; cur != null; ) {
102. last = cur;
103. dir = (p.x[cur->depth] > cur->val.x[cur->depth]);
104. cur = cur->ch[dir];
105. }
106. Node \*t = newNode(p, (last->depth + 1) % d), \*bad = null;
107. last->set(t, dir);
108. for( ; t != null; t = t->p) {
109. t->update();
110. if (!t->balanced()) bad = t;
111. }
112. if (bad != null) rebuild(bad);
113. }
114. long long calcEval(Point u, Node \*t, int d) {//计算估价函数
115. long long l = t->minv.x[d], r = t->maxv.x[d], x = u.x[d];
116. if (x >= l && x <= r) return 0LL;
117. long long ret = min(abs(x - l), abs(x - r));
118. return ret \* ret;
119. }
120. void updateAns(Point u, Point p) {
121. // 在这里更新答案
122. // u是kdtree的节点，P是询问的坐标
123. // 求第ask\_m近
124. long long dis = u-p;
125. if (HEAP.size() < ask\_m) //堆是否有ask\_m个元素
126. {
127. HEAP.push(make\_pair(dis, u));//如果没有，那么进入新元素
128. }else //如果已经有ask\_m个元素了，那么距离P最远的节点出堆，加入新的元素
129. {
130. if (HEAP.top().first > dis)
131. {
132. HEAP.pop();
133. HEAP.push(make\_pair(dis, u));
134. ret = HEAP.top().first;//ret为距离P第ask\_m近的元素的距离
135. }
136. }
137. }
138. void query(Node \*t, Point p) {
139. if (t == null) return;
140. updateAns(t->val, p); //遍历到一个节点，去update一下堆里的元素
141. long long evalLeft = calcEval(p, t->ch[0], t->depth);//估算该节点左儿子的距离
142. long long evalRight = calcEval(p, t->ch[1], t->depth);//估算该节点右儿子的距离
143. if (evalLeft <= evalRight) {//简单的确定回溯先后的顺序,先遍历小的,贪心而已
144. //估价已经偏小了，如果ret依旧比估价小，说明答案不在这个分支里
145. //所以这里必须满足ret>估价，才能进行回溯搜索
146. if (ret > evalLeft) query(t->ch[0], p);
147. if (ret > evalRight) query(t->ch[1], p);
148. } else {
149. if (ret > evalRight) query(t->ch[1], p);
150. if (ret > evalLeft) query(t->ch[0], p);
151. }
152. }
153. void query(Point p) {
154. query(root, p); //去计算求解
155. //下面的代码，就是输出堆中的元素而已。
156. Point pt[20];
157. for (int j = 0; !HEAP.empty(); ++ j)
158. {
159. pt[j] = HEAP.top().second;
160. HEAP.pop();
161. }
162. for (int j = ask\_m-1;j>=0;--j) print(pt[j]);
163. }
164. };
165. void initNull() {
166. totNode = nodePool;
167. null = totNode ++;
168. null->size = 0;
169. for(int i = 0; i < d; ++ i) {//初始化第一个节点
170. null->maxv.x[i] = -INF;
171. null->minv.x[i] = INF;
172. }
173. }
174. KDTree kdt;
175. int main()
176. {
177. int n;
178. while (~scanf("%d%d", &n, &d))
179. {
180. for (int i = 0; i < n; ++ i)
181. for (int j = 0; j < d; ++ j)
182. scanf("%d", &buf[i].x[j]);
183. initNull();
184. kdt = KDTree(buf, n/4);
185. for (int i = n/4; i < n; ++ i)
186. kdt.insert(Point({buf[i]}));
187. int T;
188. scanf("%d", &T);
189. while ( T -- )
190. {
191. Point tmp;
192. for (int i = 0; i < d; ++ i)
193. scanf("%d", &tmp.x[i]);
194. scanf("%d", &ask\_m);
195. printf("the closest %d points are:\n", ask\_m);
196. ret=100000000000000LL;
197. kdt.query(tmp);
198. }
199. }
200. }

## KMP与扩展KMP与马拉车算法

### KMP：普通KMP

1、所谓KMP算法，首先通读模板

KMP模板

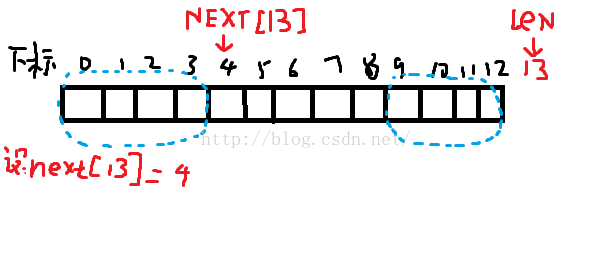
1. //pattern为模板串，从0下标，长度为len。 返回next数组
2. template<typename T>
3. void kmp\_pre(T pattern[], int len, int next[])
4. {
5. next[0] = next[1] = 0;
6. for(int i = 1 ; i < len ; i++)
7. {
8. int j = next[i];
9. while(j && pattern[i] != pattern[j])
10. j = next[j];
11. next[i+1] = pattern[i] == pattern[j] ? j+1 : 0;
12. }
13. }
14. //text为匹配串，lenT为其长度。pattern为模板串，lenP为其长度，next为上面得到的next数组。
15. //返回一个vector，表示所有匹配成功的在text的下标（从0开始）
16. //还返回一个匹配成功的数量
17. vector<int>ret;
18. template<typename T>
19. bool find(T text[], int lenT, T pattern[], int lenP, int next[], vector<int> &ret)//下标皆为从0开始
20. {
21. ret.clear();
22. int j = 0; //初始化在模式串第一个位置
23. for (int i = 0; i < lenT; ++ i)
24. {
25. while (j && pattern[j] != text[i]) j = next[j];
26. if (pattern[j] == text[i]) j++;
27. if (j == lenP)
28. {
29. ret.push\_back(i-lenP+1);
30. }
31. }
32. return ret.size();
33. }

2、next数组含义

KMP算法最难的地方，就是next数组的含义。

下面的叙述中，所有的下标都是从0开始。

next[i]是串的，和串的部分是一模一样的。



当然，对于i=len的情况，也是一样的。

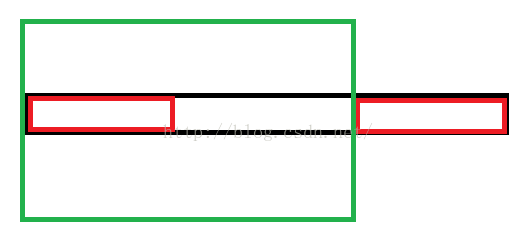
3、求字符串最少添加几个字符，可以形成有循环节的字符串

（求原字符串，循环节最大是多长）

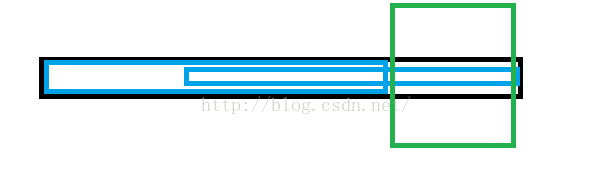
比如，a，添加一个a变为aa，循环节为a。

abcab添加一个c，变为[abc][abc]循环节为abc

abcxxxa 添加bcxxx变为abcxxxabcxxx，循环节为abcxxx

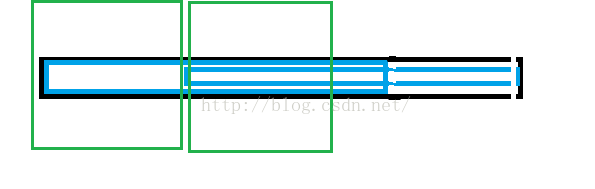


显然，对于next[len]，如果相同的两串字符串和上图一样是红色的部分，那么循环节一定是绿色的部分。（原因是，如果有更短的，无论如何，最后红色的部分一定有一部分是循环节的一部分。如果循环节更短的话，（右边的绿色向左边移动一些），那么右边红色部分作为一个循环的起点显然是不够的…… 所以这就是最小的循环节了。）

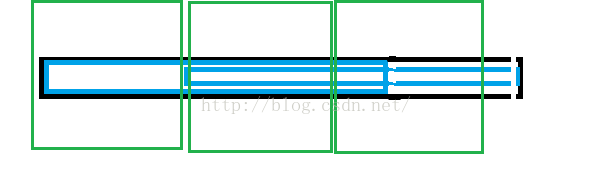


如果利用next[i]得到的两个相同的区域，是上图蓝色的区域（有重叠部分），那么循环节一定是绿色的部分。

首先简单说明为什么这样是对的。



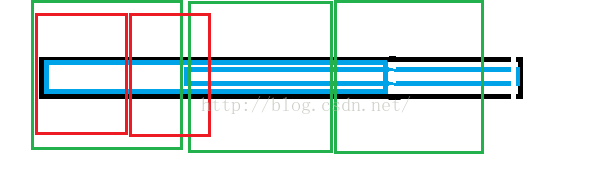
左边两块绿色是完全相同的（因为两个串的首部一定是相同的）。然后第二个绿色方框，不仅是第二个串的前缀，还是第一个串的一部分。把两个方框看为一个整体，视为第一个串的一部分的话。可以得到



三个绿色方框的字符都是相同的！最终可以证明出，绿色方框是循环串的一部分。

但是为啥绿色就是最小的？

假设绿色方框变小一点点……变为下图红色的那样。



至少说，我们不能保证这2个红色串是相同的……

【假设】两个红色部分是相同的，假设这是一个合法情况，那么会出现什么情况呢？

对，情况就是，蓝色部分一定是画错了…… 第二个蓝色部分的开始位置一定是图中2个红色方框结合的位置……

所以我们可以得到一个循环节长度的公式：

cir = len - next[len]   (公式由上2个图得到，化简后结果为这个。一个式子满足上述2个图的情况，所有下标为从0开始。 KMP程序采用前面我的程序模板)

特判1：整个串为 abcd的情况（没有重复部分）

特判2：整个串为a一个字母的情况(需要整体复制一份……)

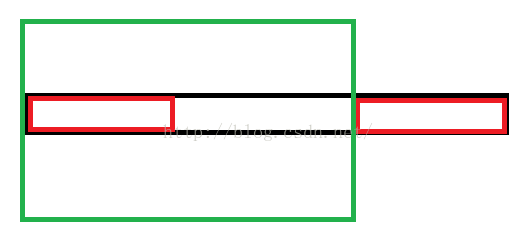
然后这题就做完了。

4、有哪些p，可以对任意i都可以让s[i] == s[i+p] （i+p不超字符串范围）

等价于问：有哪些长度的循环节，对字符串均成立。z

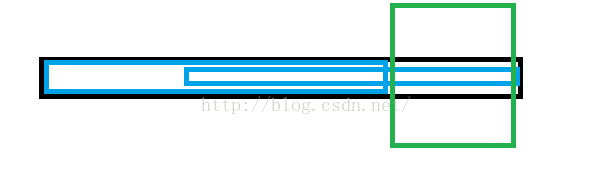
对于任何一个可行的循环节，必定为串首位部分有相同的地方。依旧用这个图。如图所示。

情况1：如图



图中，红色的地方是相同的串，只有是这样，绿色部分是循环节。

情况2：如图



图中，蓝色的部分是相同的串，绿色部分就是循环节。

任何一个可行的循环节，必定是上述两个情况之一，所以只需要考虑所有与相等的情况(是串长的意思)。通俗的说，就是

,显然与(就是串的长度为的后缀)是相同的(数组的含义)。然后继续求,这依旧保证与是相同的，一直循环下去，直到,表示没有和串的后缀相同的。对于每一个,都存在一个长度的串，可以是整个串的循环节。当然，的时候，公式一样成立（整个串是一个循环节）。

5、一个串，由子串重复得到相关。

这种问题，实际上还是等价于求循环节问题。

### EXKMP：扩展KMP

1. 扩展KMP代码的介绍与使用
2. //得到扩展KMP的next数组。next[i]，T从i开始的串，和pattern的前缀公共长度
3. //对不求extend[i]的数组，首先进行使用。
4. template<typename T>
5. void get\_extand\_next(T pattern[], int lenP, int next[])
6. {
7. int a(0);
8. next[0] = lenP;
9. for (; a != lenP && pattern[a] == pattern[a + 1]; ++ a);
10. next[1] = a;
11. a = 1;
12. for (int k = 2; k < lenP; ++ k)
13. {
14. int p = a + next[a] - 1, L = next[k - a];
15. //p是最远匹配下标，L是当前串
16. if (k-1+L >= p)
17. {
18. int j = (p - k + 1) > 0 ? p - k + 1 : 0;
19. while (k + j < lenP && pattern[k + j] == pattern[j]) j++;
20. next[k] = j;
21. a = k;
22. }
23. else next[k] = L;
24. }
25. }
26. template<typename T>
27. void get\_extend(T text[], int lenT, T pattern[], int lenP, int next[], int extend[], bool flag = 0)//flag表示是否已经得到next数组了,0表示否
28. //extend[i]表示text[i..lenT-1]和pattern[0..lenP-1]串的公共前缀
29. {
30. if (!flag) get\_extand\_next(pattern, lenP, next);
31. int a = 0;
32. int len = min(lenT, lenP);//两个串取短
33. while (a < len && text[a] == pattern[a]) a++;
34. extend[0] = a;
35. a = 0;
36. for (int k = 1; k != lenT; k ++ )
37. {
38. int p = a + extend[a] - 1, L = next[k - a];
39. if (k - 1 + L >= p)
40. {
41. int j = (p - k + 1) > 0 ? p - k + 1 : 0;
42. while (k + j < lenT && j < lenP && text[k + j] == pattern[j]) j++;
43. extend[k] = j;
44. a = k;
45. }
46. else extend[k] = L;
47. }
48. }

2、扩展KMP的应用

扩展KMP的应用面比较窄，通常为“最长回文子串”，与“最长重复子串”这两种。其中，最长回文子串速度很慢，不如malache算法。使用扩展KMP，往往要结合二分与归并的思想来做题。

### Manacher算法介绍

1. 代码与模板

//获得manacher算法数组

//给定一个text模板串，下标从0开始。长度为len，输出一个malache数组。同时需要提供一个运算辅助数组str，其类型与text相同

//算法流程：首先给字符首插入未出现的字符，第奇数个字符开始插入#等相同的，并且原字符串没有出现的字符。

//然后在第偶数个位置，按顺序插入原串。比如abbacabba会变为^#a#b#b#a#c#a#b#b#a\* 这样的串，首尾都是未出现的串。这样是为了程序避免越界，为了程序好写。

//然后会返回一个manacher数组，表示对于从^#a#b#b#a#c#a#b#b#a\*这样的字符串中，每个下标为中心，向左右扩展，能得到的最长回文串的长度。

//实质上在malache串的pos位置，就是在原串的(pos-malache[pos]+2)/2-1为左端点，长度为malache[pos]-1的回文串。

1. #include <typeinfo>
2. template<typename T>
3. void manacher(T text[], int len, int malache[], T str[])
4. {
5. T inf, space;
6. if (typeid(T).name() == typeid(char).name())
7. {
8. inf= -128;
9. space = 2 - 127;
10. }
11. if (typeid(T).name() == typeid(int).name())
12. {
13. inf= - 0x3f3f3f3f;
14. space = 2 -0x3f3f3f3f;
15. }
16. int l = 0;
17. str[l++] = inf;
18. str[l++] = space;
19. for (int i = 0; i < len; ++ i)
20. {
21. str[l++] = text[i];
22. str[l++] = space;
23. }
24. str[l] = inf + 1;
25. int mx=0, id=0;//mx:最远匹配距离,id:最远距离对应的坐标的下标
26. for (int i = 0; i < l; ++ i)
27. {
28. malache[i] = mx > i ? min(malache[2 \* id - i], mx - i) : 1;
29. while (str[i + malache[i]] == str[i - malache[i]]) malache[i] ++ ;
30. if (i + malache[i] > mx)
31. {
32. mx = i + malache[i];
33. id = i;
34. }
35. }
37. }
38. 马拉车算法的应用

貌似只能求最长回文。但是思想很值得借鉴而已。

## 组合游戏与SG函数

### SG函数

如果一个游戏有若干个状态，每次经过选择，可以进入到下一个状态。如果没后后继状态，则为输。

如果对于状态P，他没有后继装填，那么SG(P)=0。否则，SG(P)为P的所有后继状态（比如a,b,c,d,e,f,g）中SG函数值没出现过的值。

比如sg(a)=1,sg(b)=2,sg(c)=3,sg(d)=0,sg(e)=15,sg(f)=10,sg(g)=11,那么sg(P)=4，因为0,1,2,3,10,11,15中间，最先缺少的数字是4。

### 问题组合

如果题目给出若干个状态，每次可以改变其中一个状态，我们只需要求出所有状态的sg函数值，然后xor起来所有的数值。如果为0，则为先手必输，否则为赢。

## 后缀数组

### 板子

（后缀：后缀是指从某个位置i开始到整个串末尾结束的一个特殊子串。）

#### SuffixArray（O(nlogn)的倍增法后缀数组）

1. const int maxn = 1e6 + 10;
2. const int maxlog = 20;
3. struct Suffix\_array {
4. //调用方法：sa[i]表示排名第i的后缀是哪一个
5. //使用之前init(n, s) n为字符串长度 和 字符串
6. //get\_sa(int m) 传入字符种类数m 获得后缀数组sa
7. //get\_height() 获得height数组和rank数组
8. //height[i]表示sa[i]和sa[i-1]的最长公共前缀 rank[i]表示后缀i的排名
9. //init\_RMQ() 初始化查询最长公共前缀
10. //query(int a, int b) 获得后缀a和后缀b的最长公共前缀
11. //
12. //warning:
13. //所有函数调用必须要从上到下
14. //maxn为字符串长度
15. //字符串结尾为0 其他均大于0 sa[0]为n-1 rank[n-1]为0
16. //数组下标从0开始
17. int sa[maxn], \*s, n;//空间要三倍的n
18. int rank[maxn], height[maxn];
19. int t[maxn], t2[maxn], c[maxn]; // 辅助数组
20. int d[maxn][maxlog];//从i开始,长度为2^j的最值 查询最长公共前缀用
21. void init(int n, int\* s) { //n为字符串长度 和 字符串
22. this->n = n; this->s = s;
23. // memset(d, 0, sizeof(d));
24. }
25. void get\_sa(int m) {
26. // m为最大字符值加1（一般情况是'z'+1）。调用之前需设置好s和n
27. int i, \*x = t, \*y = t2, p, k;
28. for (i = 0; i < m; i++) c[i] = 0;
29. for (i = 0; i < n; i++) c[x[i] = s[i]]++;
30. for (i = 1; i < m; i++) c[i] += c[i - 1];
31. for (i = n - 1; i >= 0; i--) sa[--c[x[i]]] = i;
32. for (k = 1, p = 1; k <= n && p < n; k <<= 1, m = p) {
33. for (p = 0, i = n - k; i < n; i++) y[p++] = i;
34. for (i = 0; i < n; i++) if (sa[i] >= k) y[p++] = sa[i] - k;
35. for (i = 0; i < m; i++) c[i] = 0;
36. for (i = 0; i < n; i++) c[x[y[i]]]++;
37. for (i = 1; i < m; i++) c[i] += c[i - 1];
38. for (i = n - 1; i >= 0; i--) sa[--c[x[y[i]]]] = y[i];
39. for (swap(x, y), p = 1, x[sa[0]] = 0, i = 1; i < n; i++)
40. x[sa[i]] = y[sa[i-1]]==y[sa[i]] && y[sa[i-1]+k]==y[sa[i]+k] ? p-1 : p++;
41. }
42. }
43. void get\_height() {
44. int k = 0, j;
45. for (int i = 1; i < n; i++) rank[sa[i]] = i;// rank[i]表示后缀i的排名
46. for (int i = 0; i < n - 1; height[rank[i++]] = k)
47. for (k?k--:0, j = sa[rank[i]-1]; s[i+k]==s[j+k]; k++);
48. }
49. void init\_RMQ() {
50. for (int i = 0; i < n; ++i) d[i][0] = height[i];
51. for (int j = 1; (1<<j) <= n; ++j)
52. for (int i = 0; i + (1<<j) - 1 < n; ++i)
53. d[i][j] = min(d[i][j-1], d[i+(1<<(j-1))][j-1]);
54. }
55. int query(int begin, int end) {//查询后缀begin 和 后缀end 的最长公共前缀
56. int L = rank[begin], R = rank[end];
57. if (L > R) swap(L, R); L++;
58. int k = logz(R-L+1);
59. return min(d[L][k], d[R-(1<<k)+1][k]);
60. }
61. } g;

#### SuffixArray（O(n)的dc3后缀数组）

1. const int maxn = 400 + 10;
2. const int maxlog = 10;
3. struct Suffix\_array {
4. //调用方法：sa[i]表示排名第i的后缀是哪一个
5. //使用之前init(n, s) n为字符串长度 和 字符串
6. //get\_sa(int m) 传入字符种类数m 获得后缀数组sa
7. //get\_height() 获得height数组和rank数组
8. //height[i]表示sa[i]和sa[i-1]的最长公共前缀 rank[i]表示后缀i的排名
9. //init\_RMQ() 初始化查询最长公共前缀
10. //query(int a, int b) 获得后缀a和后缀b的最长公共前缀
11. //
12. //warning:
13. //所有函数调用必须要从上到下
14. //maxn为字符串长度的4倍
15. //字符串结尾为0 其他均大于0 sa[0]为n-1 rank[n-1]为0
16. //数组下标从0开始
17. #define F(x) ((x)/3+((x)%3==1?0:tb))
18. int sa[maxn], \*s, n;//空间要三倍的n
19. int rank[maxn], height[maxn];
20. int d[maxn][maxlog];//从i开始,长度为2^j的最值 查询最长公共前缀用
21. int wa[maxn], wb[maxn], wv[maxn], ws[maxn]; //临时变量
22. void init(int n, int\* s) { this->n = n; this->s = s; } //n为字符串长度和字符串
23. void get\_sa(int m) { dc3(s, sa, n, m); } //传入字符种类数m 获得后缀数组sa
24. int c12(int k, int\* r, int a, int b) {
25. if(k==2) return r[a]<r[b]||r[a]==r[b]&&c12(1,r,a+1,b+1);
26. else return r[a]<r[b]||r[a]==r[b]&&wv[a+1]<wv[b+1];
27. }
28. void sort(int\* r, int\* a, int\* b, int n, int m) {
29. for(int i=0; i<n; i++) wv[i]=r[a[i]];
30. for(int i=0; i<m; i++) ws[i]=0;
31. for(int i=0; i<n; i++) ws[wv[i]]++;
32. for(int i=1; i<m; i++) ws[i]+=ws[i-1];
33. for(int i=n-1; i>=0; i--) b[--ws[wv[i]]]=a[i];
34. }
35. void dc3(int\* r, int\* sa, int n, int m) {
36. //字符数组s, sa, length(s), 字符种类 (从这里开始
37. int i, j, \*rn=r+n, \*san=sa+n, ta=0, tb=(n+1)/3, tbc=0,p;
38. r[n] = r[n+1] = 0;
39. for(i=0; i<n; i++) if(i%3 != 0) wa[tbc++] = i;
40. sort(r+2, wa, wb, tbc, m);
41. sort(r+1, wb, wa, tbc, m);
42. sort(r, wa, wb, tbc, m);
43. for(p=1, rn[F(wb[0])]=0, i=1; i<tbc; i++)
44. rn[F(wb[i])] = r[wb[i-1]]==r[wb[i]] && r[wb[i-1]+1]==r[wb[i]+1]
45. && r[wb[i-1]+2]==r[wb[i]+2]?p-1:p++;
46. if(p < tbc) dc3(rn, san, tbc, p);
47. else for(i=0; i<tbc; i++) san[rn[i]] = i;
48. for(i=0; i<tbc; i++) if(san[i] < tb) wb[ta++] = san[i]\*3;
49. if(n % 3 == 1) wb[ta++] = n-1;
50. sort(r, wb, wa, ta, m);
51. for(i=0; i<tbc; i++)
52. wv[wb[i]=(san[i]<tb?san[i]\*3+1:(san[i]-tb)\*3+2)]=i;
53. for(i=0, j=0, p=0; i<ta && j<tbc; p++)
54. sa[p] = c12(wb[j]%3,r,wa[i],wb[j])?wa[i++]:wb[j++];
55. for(; i<ta; p++) sa[p] = wa[i++];
56. for(; j<tbc; p++) sa[p] = wb[j++];
57. }
58. void get\_height() {
59. int k = 0, j;
60. //得到rank，rank[i]表示后缀i的排名
61. for (int i = 1; i < n; i++) rank[sa[i]] = i;
62. for (int i = 0; i < n - 1; height[rank[i++]] = k)
63. for (k?k--:0, j = sa[rank[i]-1]; s[i+k]==s[j+k]; k++);
64. }
65. void init\_RMQ() {
66. for (int i = 0; i < n; ++i) d[i][0] = height[i];
67. for (int j = 1; (1<<j) <= n; ++j)
68. for (int i = 0; i + (1<<j) - 1 < n; ++i)
69. d[i][j] = min(d[i][j-1], d[i+(1<<(j-1))][j-1]);
70. }
71. int query(int begin, int end) {//查询后缀begin 和 后缀end 的最长公共前缀
72. int L = rank[begin], R = rank[end];
73. if (L > R) swap(L, R); L++;
74. int k = logz(R-L+1);
75. return min(d[L][k], d[R-(1<<k)+1][k]);
76. }
77. } g;

### 实际效率：

测试环境：NOI-linux Pentium(R) 4 CPU 2.80GHz

|  |  |  |
| --- | --- | --- |
| N | 倍增算法 | DC3算法 |
| 200000 | 192 | 140 |
| 300000 | 367 | 244 |
| 500000 | 750 | 499 |
| 1000000 | 1693 | 1248 |

（不包括读入和输出的时间，单位：ms）

### 后缀数组的应用

例1：最长公共前缀

给定一个字符串，询问某两个后缀的最长公共前缀。

算法分析：

直接调用板子里的query(int a, int b) 获得后缀a和后缀b的最长公共前缀

例2：可重叠最长重复子串

（重复子串：字符串R在字符串L中至少出现两次，则称R是L的重复子串。）

给定一个字符串，求最长重复子串，这两个子串可以重叠。

算法分析：

这道题是后缀数组的一个简单应用。做法比较简单，只需要求height数组里的最大值即可。

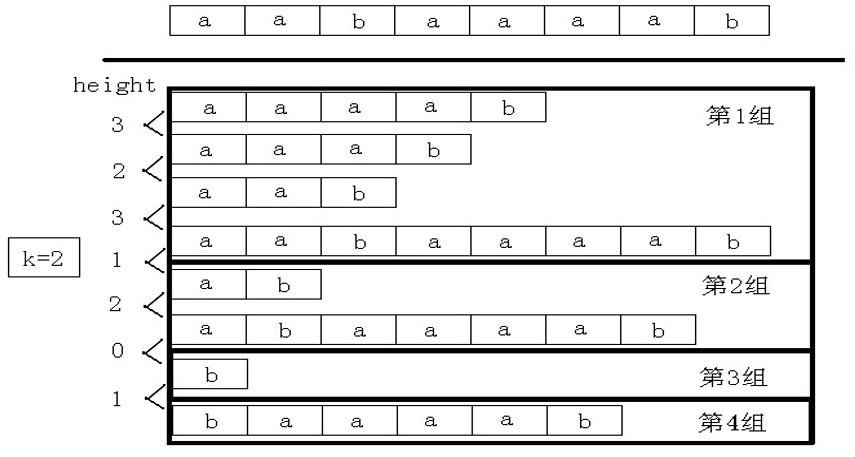
时间复杂度为O(n)。

例3：不可重叠最长重复子串（pku1743）

给定一个字符串，求最长重复子串，这两个子串不能重叠。

算法分析：

先二分答案，把题目变成判定性问题：判断是否存在两个长度为k的子串是相同的，且不重叠。解决这个问题的关键还是利用 height数组。把排序后的后缀分成若干组，其中每组的后缀之间的height值都不小于k。例如，字符串为“aabaaaab”，当k=2时，后缀分成了 4组，如图所示。



容易看出，有希望成为最长公共前缀不小于k的两个后缀一定在同一组。然后对于每组后缀，只须判断每个后缀的sa值的最大值和最小值之差是否不小于 k。如果有一组满足，则说明存在，否则不存在。整个做法的时间复杂度为 O(nlogn)。

例4：可重叠的k次最长重复子串（pku3261）

给定一个字符串，求至少出现k次的最长重复子串，这k个子串可以重叠。

算法分析：

这题的做法和上一题差不多，也是先二分答案，然后将后缀分成若干组。不同的是，这里要判断的是有没有一个组的后缀个数不小于k。如果有，那么存在 k个相同的子串满足条件，否则不存在。这个做法的时间复杂度为O(nlogn)。

例5：不相同的子串的个数（spoj694,spoj705）

给定一个字符串，求不相同的子串的个数。

算法分析：

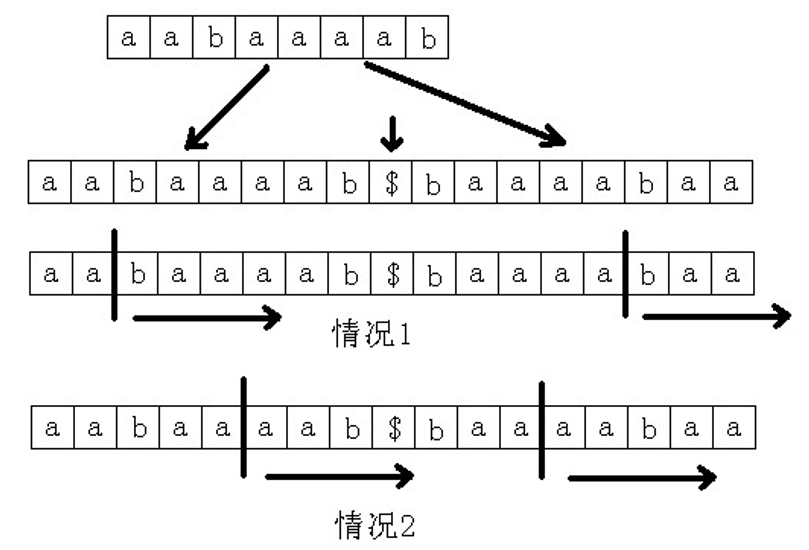
每个子串一定是某个后缀的前缀，那么原问题等价于求所有后缀之间的不相同的前缀的个数。如果所有的后缀按照 suffix(sa[1]), suffix(sa[2]), suffix(sa[3]), …… ,suffix(sa[n])的顺序计算，不难发现，对于每一次新加进来的后缀 suffix(sa[k]),它将产生 n-sa[k]+1 个新的前缀。但是其中有 height[k]个是和前面的字符串的前缀是相同的。所以suffix(sa[k])将“贡献” 出n-sa[k]+1- height[k]个不同的子串。累加后便是原问题的答案。这个做法的时间复杂度为O(n)。

例6：最长回文子串（ural1297）

给定一个字符串，求最长回文子串。

算法分析：

穷举每一位，然后计算以这个字符为中心的最长回文子串。注意这里要分两种情况，一是回文子串的长度为奇数，二是长度为偶数。两种情况都可以转化为求一个后缀和一个反过来写的后缀的最长公共前缀。具体的做法是：将整个字符串反过来写在原字符串后面，中间用一个特殊的字符隔开。这样就把问题变为了求这个新的字符串的某两个后缀的最长公共前缀。如图所示



这个做法的时间复杂度为O(nlogn)。

例7：连续重复子串(pku2406)

给定一个字符串L，已知这个字符串是由某个字符串S重复R次而得到的，求R的最大值。

算法分析：

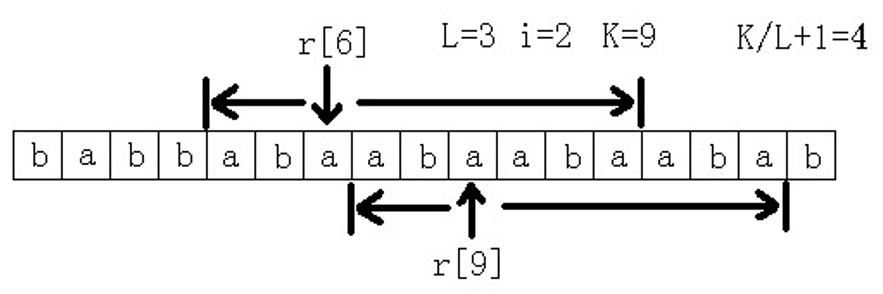
做法比较简单，穷举字符串S的长度k，然后判断是否满足。判断的时候，先看字符串L的长度能否被k整除，再看suffix(1)和suffix(k+1)的最长公共前缀是否等于n-k。在询问最长公共前缀的时候，suffix(1)是固定的，所以RMQ 问题没有必要做所有的预处理，只需求出 height 数组中的每一个数到 height[rank[1]]之间的最小值即可。整个做法的时间复杂度为O(n)。

例8：重复次数最多的连续重复子串(spoj687,pku3693)

给定一个字符串，求重复次数最多的连续重复子串。

算法分析：

先穷举长度L，然后求长度为L的子串最多能连续出现几次。首先连续出现 1次是肯定可以的，所以这里只考虑至少2次的情况。假设在原字符串中连续出现 2 次，记这个子字符串为 S，那么 S 肯定包括了字符 r[0], r[L], r[L\*2], r[L\*3], ……中的某相邻的两个。所以只须看字符r[L\*i]和r[L\*(i+1)]往前和往后各能匹配到多远，记这个总长度为K，那么这里连续出现了K/L+1次。最后看最大值是多少。如图所示。



穷举长度L的时间是n，每次计算的时间是n/L。所以整个做法的时间复杂度是O(n/1+n/2+n/3+……+n/n)=O(nlogn)。

参考代码：

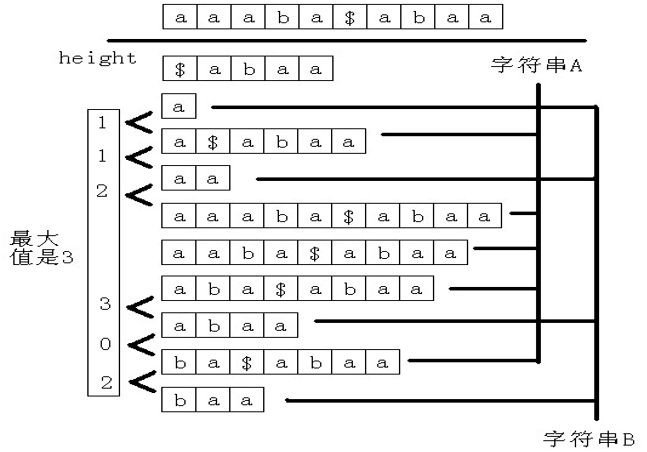
1. char tmpS[maxn];
2. int s[maxn];
3. void solve() {
4. int n = strlen(tmpS);
5. for (int i = 0; i < n; i++) {
6. s[i] = tmpS[i] - 'a' + 1;
7. }
8. s[n] = 0; n++;
9. g.init(n, s);
10. g.get\_sa(30);
11. g.get\_height();
12. g.init\_RMQ();
13. int ans = -1;//重复次数
14. vector<int> ansLen;//答案可能的循环节长度
15. bool flag;
16. int tmpLen, beg;
17. for (int L = 1; L <= n / 2; L++) { //枚举循环节长度，并假设都可以循环两次及以上
18. for (int i = 0; (i + 1) \* L < n; i++) {
19. flag = false;
20. tmpLen = g.query(i\*L, (i+1)\*L);
21. if (tmpLen % L && i) {
22. flag = true;
23. beg = i\*L - (L - tmpLen%L);
24. tmpLen = g.query(beg, beg+L);
25. }
26. if (tmpLen / L + 1 == ans) {
27. ansLen.push\_back(L);
28. }
29. if (tmpLen > 0 && tmpLen / L + 1 > ans) {
30. ans = tmpLen / L + 1;
31. ansLen.clear();
32. ansLen.push\_back(L);
33. }
34. }
35. }
36. if (ans < 0) { //当循环一次时
37. char ch = 'z';
38. for (int i = 0; i < n - 1; i++) {
39. ch = min(tmpS[i], ch);
40. }
41. printf("%c\n", ch);
42. return;
43. }
44. int ansL = INF, ansR;
45. flag = true;
47. for (int i = 0; i < ansLen.size(); i++) {
48. for (int j = 0; j + ansLen[i] <= n - 1; j++) {
49. if (g.query(j, j + ansLen[i]) / ansLen[i] + 1 == ans) {
50. if (flag || g.rank[ansL] > g.rank[j]) {
51. ansL = j;
52. ansR = ansL + ans \* ansLen[i] - 1;
53. flag = false;
54. }
55. }
56. }
57. }
58. for (int i = ansL; i <= ansR; i++) {
59. printf("%c", tmpS[i]);
60. }
61. printf("\n");
62. }
63. int main() {
64. int kase = 0;
65. while (scanf("%s", tmpS) != EOF) {
66. if (tmpS[0] == '#') break;
67. printf("Case %d: ", ++kase);
68. solve();
69. }
70. return 0;
71. }

例9：最长公共子串(pku2774,ural1517)

给定两个字符串A和B，求最长公共子串。

算法分析:

字符串的任何一个子串都是这个字符串的某个后缀的前缀。求A和B的最长公共子串等价于求 A的后缀和 B的后缀的最长公共前缀的最大值。如果枚举 A 和B的所有的后缀，那么这样做显然效率低下。由于要计算A的后缀和B的后缀的最长公共前缀，所以先将第二个字符串写在第一个字符串后面，中间用一个没有出现过的字符隔开，再求这个新的字符串的后缀数组。观察一下，看看能不能从这个新的字符串的后缀数组中找到一些规律。以A=“aaaba”，B=“abaa”为例，如图8所示。



那么是不是所有的height值中的最大值就是答案呢？不一定！有可能这两个后缀是在同一个字符串中的，所以实际上只有当 suffix(sa[i-1])和 suffix(sa[i])不是同一个字符串中的两个后缀时，height[i]才是满足条件的。而这其中的最大值就是答案。记字符串A和字符串B的长度分别为|A|和|B|。求新的字符串的后缀数组和height数组的时间是O(|A|+|B|)，然后求排名相邻但原来不在同一个字符串中的两个后缀的 height 值的最大值，时间也是 O(|A|+|B|)，所以整个做法的时间复杂度为 O(|A|+|B|)。时间复杂度已经取到下限，由此看出，这是一个非常优秀的算法。

例10:长度不小于k的公共子串的个数(pku3415)

给定两个字符串A和B，求长度不小于k的公共子串的个数（可以相同）。

样例1:

A=“xx”，B=“xx”，k=1，长度不小于k的公共子串的个数是5。

样例2:

A=“aababaa”，B=“abaabaa”，k=2，长度不小于k的公共子串的个数是22。

算法分析:

基本思路是计算A的所有后缀和B的所有后缀之间的最长公共前缀的长度，把最长公共前缀长度不小于k的部分全部加起来。先将两个字符串连起来，中间用一个没有出现过的字符隔开。按height值分组后，接下来的工作便是快速的统计每组中后缀之间的最长公共前缀之和。扫描一遍，每遇到一个B的后缀就统计与前面的A的后缀能产生多少个长度不小于k的公共子串，这里A的后缀需要用一个单调的栈来高效的维护。然后对A也这样做一次。

参考代码：

1. const int maxl = 100000 + 10;
2. char str1[maxl], str2[maxl];
3. typedef pair<LL, LL> pll;
4. int s[maxn];
5. pll ss[maxn];//该元素，该元素的个数
6. int len1, len2, same;
7. void solve() {
8. len1 = strlen(str1);
9. len2 = strlen(str2);
10. int minLen = min(len1, len2);
11. for (int i = 0; i < len1; i++) {
12. s[i] = str1[i];
13. }
14. s[len1] = 10;
15. for (int i = 0; i < len2; i++) {
16. s[i + len1 + 1] = str2[i];
17. }
18. s[len1 + len2 + 1] = 0;
19. int n = len1 + len2 + 2;
20. g.init(n, s);
21. g.get\_sa(200);
22. g.get\_height();
23. LL tot = 0, top = 1;
24. LL ans = 0;
25. ss[0] = pll(0, 0);
26. for (int i = 0; i < n; i++) {
27. if (g.height[i] < same) {
28. tot = 0;
29. top = 1;
30. continue;
31. }
32. int cnt = 0;
33. if (g.sa[i - 1] < len1) {
34. tot += g.height[i] - same + 1;
35. cnt++;
36. }
37. while (g.height[i] <= ss[top - 1].first) {
38. top--;
39. tot -= ss[top].second \* (ss[top].first - g.height[i]);
40. cnt += ss[top].second;
41. }
42. ss[top++] = pll(g.height[i], cnt);
43. if (g.sa[i] > len1) {
44. ans += tot;
45. }
46. }
47. tot = 0;
48. top = 1;
49. for (int i = 0; i < n; i++) {
50. if (g.height[i] < same) {
51. tot = 0;
52. top = 1;
53. continue;
54. }
55. int cnt = 0;
56. if (g.sa[i - 1] > len1) {
57. tot += g.height[i] - same + 1;
58. cnt++;
59. }
60. while (g.height[i] <= ss[top - 1].first) {
61. top--;
62. tot -= ss[top].second \* (ss[top].first - g.height[i]);
63. cnt += ss[top].second;
64. }
65. ss[top++] = pll(g.height[i], cnt);
66. if (g.sa[i] < len1) {
67. ans += tot;
68. }
69. }
70. printf("%lld\n", ans);
71. }
72. int main() {
73. while (scanf("%d", &same) == 1 && same) {
74. scanf("%s%s", str1, str2);
75. solve();
76. }
77. return 0;
78. }

例11:不小于k个字符串中的最长子串(pku3294)

给定n个字符串，求出现在不小于k个字符串中的最长子串。

算法分析:

将n个字符串连起来，中间用不相同的且没有出现在字符串中的字符隔开，求后缀数组。然后二分答案，用和例3同样的方法将后缀分成若干组，判断每组的后缀是否出现在不小于k个的原串中。这个做法的时间复杂度为O(nlogn)。

例12:每个字符串至少出现两次且不重叠的最长子串(spoj220)

给定n个字符串，求在每个字符串中至少出现两次且不重叠的最长子串。

算法分析:

做法和上题大同小异，也是先将n个字符串连起来，中间用不相同的且没有出现在字符串中的字符隔开，求后缀数组。然后二分答案，再将后缀分组。判断的时候，要看是否有一组后缀在每个原来的字符串中至少出现两次，并且在每个原来的字符串中，后缀的起始位置的最大值与最小值之差是否不小于当前答案

（判断能否做到不重叠，如果题目中没有不重叠的要求，那么不用做此判断）。

这个做法的时间复杂度为O(nlogn)。

例13:出现或反转后出现在每个字符串中的最长子串(PKU3294)

给定n个字符串，求出现或反转后出现在每个字符串中的最长子串。

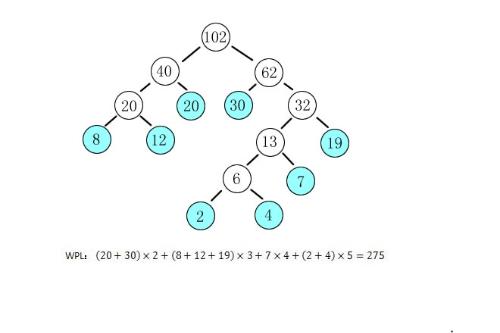
算法分析:

这题不同的地方在于要判断是否在反转后的字符串中出现。其实这并没有加大题目的难度。只需要先将每个字符串都反过来写一遍，中间用一个互不相同的且没有出现在字符串中的字符隔开，再将n个字符串全部连起来，中间也是用一个互不相同的且没有出现在字符串中的字符隔开，求后缀数组。然后二分答案，再将后缀分组。判断的时候，要看是否有一组后缀在每个原来的字符串或反转后的字符串中出现。这个做法的时间复杂度为O(nlogn)。

## 哈弗曼树

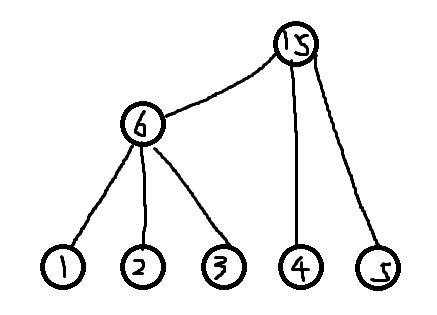
### 一、哈弗曼树介绍

所谓哈弗曼树，就是从叶节点（有权重）开始，每个权重都是儿子节点权重的和。比如下图，蓝色的是原始的节点（叶），白色的则为叶节点的父亲，哈弗曼树构造的是一颗根节点权重最小的树。

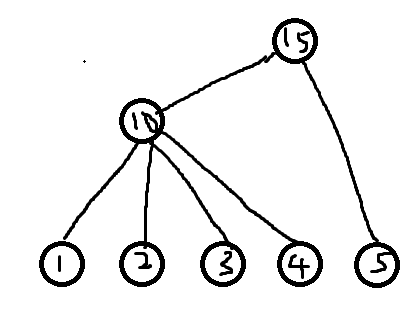


### 二、多叉哈夫曼树

如下图所示。为3叉，5个叶节点的哈弗曼树。



“满-哈弗曼树”看为上图这种哈弗曼树，三叉，并且每一个节点（非叶节点）都有3个儿子。下图则不是一颗“满-哈弗曼树”



可以看出，上面的三叉哈弗曼树，他的代价之和为6+15=21，而四岔哈弗曼树却为10+15=25。

哈弗曼树的构造，往往都是利用双队列或者堆来构造，每次选择最小的m个节点，构造出一个新节点。（注意，新节点，也可以作为未来选择的节点的待选节点）

### 三、多叉哈弗曼树公式

假设哈弗曼树为m叉，假设叶节点为x个，度为m的节点有y个，则满足公式。

### 四、多叉哈弗曼树构造法

直接把最小的m个数字合并，进行构造哈弗曼树，不一定是最优的。（上面的4叉和3叉，3叉更优。3叉也是一个数的4叉，只不过没有满而已。）

只需要补充若干个0，然后继续使用“把最小的m个数字合并”的方法，就可以构造出最优哈弗曼树了。添加个0就行了。

### 五、实现算法

1、用堆。O(nlogn)

2、把原来的数列排序，然后借助deque来缩小代码量。O(n)

## 行列式与生成树计数

### 行列式

#### 1、简介

行列式就是类似矩阵一样的一个数列，行列是有值的。行列式的值的数学定义大致如下，

每一个取一个数字，假如第一行取第三个数字，那么记为3，第二行取第5个数字，那么记为5…… 假如有n行m列，那么实际上就是m！个不同的方案（同一列只能被取一次）。然后求解，就是选出的m！个方案，每个方案选出的数字的乘积。假设P是刚才序列的逆序对的数，那么刚才算出的乘积还需要再乘上。

#### 2、举例

举个例子，求下面行列式的值。



那么3的全排列有如下几种

123，132，213，231，312，321

123的逆序对为0个，最终结果为

132的逆序对为1个，最终结果为

213的逆序对为1个，最终结果为

231的逆序对为2个，最终结果为

312的逆序对为2个，最终结果为

321的逆序对为3个，最终结果为

把上述所有结果加起来求和为0。所以上述行列式的结果为0。

#### 3、行列式的其他求法

可以利用消元的方法来求行列式。利用行列式的性质(g)行列式的某一行（列）的各元素乘同一个数字然后加到另一行(列)对应的元素上去，行列式不变。可以转化为类似高斯消元的方法来进行求解。

#### 4、行列式的性质

a)行列式的转置，值不变

b)对换行列式的两行（列），行列式变号

c)如果有两行（列）完全相同，行列式为0

d)行列式的某一行（列）中所有元素都乘上同一个数k，等于用数k乘此行列式。

e)行列式重如果两行（列）成比例，那么此行列式为0

f)如果行列式某一行（列）的元素都是两数之和，例如第i行的元素都是两数之和，那么可以拆为两个行列式之和。



g)行列式的某一行（列）的各元素乘同一个数字然后加到另一行(列)对应的元素上去，行列式不变。

### 生成树计数

#### 1、Cayley公式

完全图有颗生成树。

#### 2、生成树如何计数（Matrix-Tree 定理）

这种问题的本质，就是给你一张图，问你有多少种连通的方式，可以形成一棵生成树。解法如下。

int D[maxn][maxn];//D[i][i]为节点i的度数，其他为0

int G[maxn][maxn];// 节点a与b有边相连的话，G[a][b]=1,其他为0

C=D-G，然后求C矩阵n-1阶主子式的解。

所谓n-1阶主子式，就是去掉原来的i行i列的所有元素后的行列式的值。

#### 例题选讲

SPOJ HIGH

题目大意：求一张图生成树的个数。做法直接使用Matrix-Tree 定理即可AC。

2014上海区域赛A题

题目大意：求一张图生成树的个数，但是其中有一些边（这些边不会构成环）必须选。

做法：把给定的边的点进行缩点，形成新的点。然后套用Matrix-Tree 定理即可AC。

#### 附录

1. SPOJ HIGH的代码

#include <bits/stdc++.h>

using namespace std;

#define pr(x) x

#define prln(x) x

//#define pr(x) cout<< #x << " = " <<x<<" "

//#define prln(x) cout<< #x << " = " <<x<<endl

const int maxn = 20;

const int mat\_size = maxn;

int n, m;

template<typename T>

struct Matrix {

T a[mat\_size][mat\_size];

int x, y;//长宽

Matrix() { memset(a,0,sizeof(a)); } //返回0矩阵

Matrix(int x, int y) : x(x), y(y) { memset(a,0,sizeof(a)); }//返回0矩阵，并且x,y赋值

Matrix(int n) : x(n), y(n) { //返回n\*n的单位矩阵

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

for (int i = 0; i < n; i++) a[i][i]=1;

}

T \* operator [] (int c) { return a[c]; }

Matrix operator \* (const Matrix& rhs) {//矩阵乘法

Matrix ret;

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

for (int j = 0; j < rhs.y; j++)

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

ret.a[i][j] = (ret.a[i][j] + a[i][k] \* rhs.a[k][j] /\*% MOD\*/)/\* % MOD\*/;

ret.x = x;

ret.y = rhs.y;

return ret;

}

Matrix operator ^ (long long b) {//矩阵A的b次方

Matrix A(\*this), ret(x);

while (b) {

if(b & 1) ret = ret \* A;

b >>= 1; A = A \* A;

}

return ret;

}

Matrix operator & (long long b) {//A^0 + A^1+A^2+A^3+++A^n，其中A是矩阵。最后返回的就是一个矩阵

Matrix ret(\*this);

for (int i = ret.x; i < ret.x \* 2; i++) {

ret.a[i-ret.x][i]= 1;

ret.a[i][i] = 1;

}

ret.x <<= 1; ret.y <<= 1;

ret = ret^b;

ret.x >>= 1; ret.y >>= 1;

for (int i = 0; i < ret.x; i++)

for (int j = 0; j < ret.y; j++)

ret.a[i][j] += ret.a[i][j + ret.x];

return ret;

}

T b[maxn][maxn];

#define zero(x)((x>0? x:-x)<1e-15)

T det() //假设为x\*x的行列式，求行列式的值

{

double ret = 1, t;

int sign = 0, n = x;

memmove(b, a, sizeof(b));

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

{

if(zero(b[i][i]))

{

int j;

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

if (!zero(b[j][i])) break;

if (j == n) return 0;

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

t = b[i][k], b[i][k] = b[j][k], b[j][k] = t;

sign++;

}

ret\*= b[i][i];

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

b[i][k] /= b[i][i];

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

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

b[j][k] -= b[j][i] \* b[i][k];

}

if (sign & 1)

ret = -ret;

return ret;

}

#undef zero

void pg() {

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

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

cout<<a[i][j]<<" ";

cout<<endl;

}

cout<<endl;

}

} ;

Matrix<double>mat;

int D[maxn][maxn];//D[i][i]为i的度数，其他为0

int G[maxn][maxn];// a,b有边的话，G[a][b]=1,其他为0

void init()

{

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

mat.x=n - 1;

mat.y=n - 1;

memset(D,0,sizeof(D));

memset(G,0,sizeof(G));

while (m--)

{

int a,b;

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

--a;

--b;

D[a][a]++;

D[b][b] ++;

G[a][b] = G[b][a] = 1;

}

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

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

mat[i][j] = D[i][j] - G[i][j];

printf("%lld\n",(long long)(mat.det()+0.5));

}

int main()

{

int T;

scanf("%d", &T);

while (T--)

{

init();

}

return 0;

}

#### 模意义下的行列式求解板子

const int N = 300;

LL mat[N][N];

LL Det (int n, int mod) //按列化为下三角

{

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

{

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

{

mat[i][j] %= mod;

}

}

LL res = 1;

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

{

if (!mat[i][i])

{

bool flag = false;

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

{

if (mat[j][i])

{

flag = true;

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

{

swap (mat[i][k], mat[j][k]);

}

res = -res;

break;

}

}

if (!flag)

{

return 0;

}

}

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

{

while (mat[j][i])

{

LL t = mat[i][i] / mat[j][i];

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

{

mat[i][k] = (mat[i][k] - t \* mat[j][k]) % mod;

swap (mat[i][k], mat[j][k]);

}

res = -res;

}

}

res = (res \* mat[i][i]) % mod;

}

return (res + mod) % mod;

}

## 树

### 树的直径，树的最长链

所谓树的直径（最长链），就是树上最长的路径。如果树的边上有权，就是树的最远点对，树边没权，可以看为边权为1。树的直径和最长链的求解方法可以dp或者bfs。

bfs：随便从一个点出发，找出距离这个点最远的点st。然后从st出发，继续bfs找出距离st最远的点ed。 这样st到ed的路径，就是树的最长链。

dp：f[i][0]表示i作为树的根，他的最长儿子链的长度。f[i][0]为i为树根，他的第二长的儿子链长度。 F[i][0]与f[i][1]的求解，则只需要考虑i的儿子的情况，转移即可。

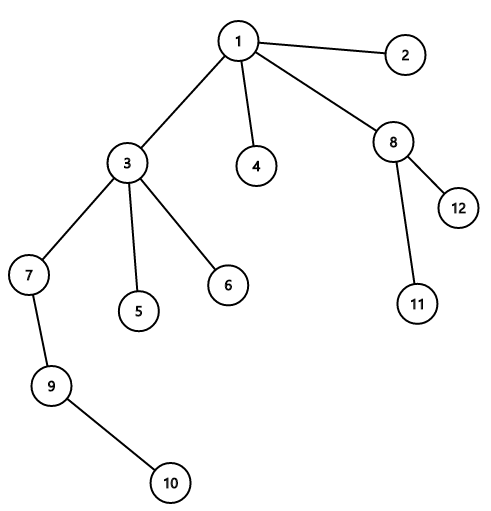
DP代码如下：

1. void dp(int now, int fa, int ans[])
2. //当前以now为根，父亲是fa，ans[i]表示以i为根的子树的最长链
3. {
4. for (auto i : g[now])
5. {
6. int will = i.will;
7. int cost = i.cost;
8. if (will == fa) continue;
9. dp(will, now, ans);
10. ans[now] = max(ans[now], ans[will]);
11. if (f[now][0] <= f[will][0] + cost)
12. {
13. f[now][1] = f[now][0];
14. f[now][0] = f[will][0] + cost;
15. }
16. else if (f[now][1] <= f[will][0] + cost)
17. {
18. f[now][1] = f[will][0] + cost;
19. }
20. if (ans[now] < f[now][1] + f[now][0])
21. ans[now] = f[now][1] + f[now][0];
22. }
23. }

## 树链剖分

### 简介

树链剖分的主要功能，是把一棵树序列化，这样便于用线段树维护。

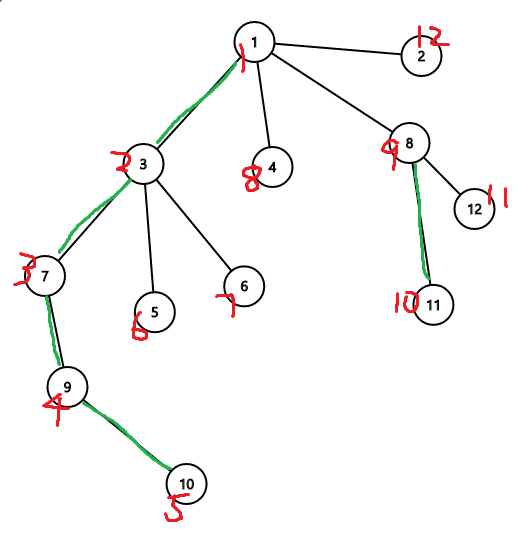


如上图所示，假设以节点1位根节点。树剖需要通过DFS得到以下一些核心参数

|  |  |
| --- | --- |
| u.size | u为根的节点数量 |
| u.dep | u在树中的深度 |
| u.son | u的中重儿子 |
| u.fa | u的父亲 |
| u.top | u所在链的顶端 |
| u.ti | u对应的线段树节点 |

所谓“重”，就是儿子多的意思。重儿子，就是儿子多的节点。比如还是以上图为例。1的重儿子是3，因为3有6个节点。3的重儿子是7，因为7的节点最多有3个（包含7）。

既然有了重儿子，和重链，那么只要每一条重链的标号都是连续的，比如上图所示，我们利用DFS序来进行标号。



如上图所示，绿色的为重链，top也就是一条重链的顶端。比如原图中9号节点的top就是1号节点（都在一条重链上，1为9所在重链的顶端）。绿色的一条链上所有的节点的“新编号”ti，也都是连续的。这样，对于查询一棵树上某个节点到某个节点的路径，就可以转化为若干个连续序列。若干个连续序列又映射在线段树上，这样树上的很多问题就很好操作和处理了。

第一步：进行DFS得到得到size,dep,fa以及son的数据。

第二步：第二次DFS，利用第一步DFS的基础（知道哪一个节点是重儿子，这个是关键），可以控制DFS顺序，从而进行连续标号。比如，以上图为例，给节点1标号为1以后，就去给3号节点标号为2，接着去给7号节点标号为3…… （因为优先给重儿子先标号，这样可以保证重链的顺序是连续的）。

第三步：构建线段树，来维护上述信息，每个节点或者边的信息，要构建好树后，后update进线段树。所以，一开始树的信息要保存下来，这个时候for一下，来update。

### 部分代码细节讲解

1、建树部分

1. struct Vector\_edge//存vector用的东西
2. {
3. int will, w;//v为连接的点， w为边权
4. Vector\_edge():will(0),w(0){}
5. Vector\_edge(int a, int b):will(a),w(b){}
6. };
7. vector<Vector\_edge>g[maxn];
8. inline void add\_edge(int a, int b, int c)
9. {
10. g[a].push\_back(Vector\_edge(b,c));
11. }

这部分代码，是一开始读入一棵树的一些信息。这棵树的边有边权。add\_edge函数，则表示a->b添加一条边，边权为c。当然实际使用的时候，要分别添加2次(a->b,b->a两次)。因为树通常是双向边。

2、两次DFS得到第一个表格中的数据。

int dfs\_clock;//DFS时间序列

struct node

{

int size, dep, son, top, fa, ti;

node():size(0),dep(0),son(0),top(0),fa(0),ti(0){};

//size:v为跟节点数量

//dep:v的深度 根为1

//top:v所在链的顶端

//son:vv在同一重链的儿子节点，重儿子

//ti:v与父亲的连边，在线段树中的位置

}t[maxn];

void dfs\_find(int u, int pa, int depth)//第一次DFS，得到size,dep,fa以及son的数据

{

t[u].son = 0; //重儿子，为0表示没有重儿子

t[u].size = 1; //节点数量

t[u].dep = depth;

t[u].fa = pa;

for (int i = 0; i != g[u].size(); ++ i)

{

int will = g[u][i].will;

if (will == pa) continue;

dfs\_find(will, u, depth + 1);

t[u].size += t[will].size;

if (t[will].size > t[t[u].son].size) t[u].son = will;

}

}

dfs\_clock就是重新编号用的。新的编号，节点下标分别从1开始往后编（不用0下标，是为了避免线段树出奇怪问题）。记得使用之前要清0.

maxn为这棵树的节点总数。dfs\_find为得到size,dep,fa以及son的数据。比较简单，不做过多介绍。

void dfs\_time(int u, int pa) // 得到时间戳等数据,u为当前节点，pa为父链顶端节点

{

t[u].ti = ++ dfs\_clock; //u这个节点的时间戳是dfs\_clock,dfs\_clock下标是从1开始的，没有0！

t[u].top = pa; //top是u所在链的顶端

if (t[u].son != 0) dfs\_time(t[u].son, t[u].top); //如果节点有重儿子，那么依旧是以pa为链顶端的一条链

for (int i = 0; i != g[u].size(); ++ i)

{

int will = g[u][i].will;

if (will == t[u].son || will == t[u].fa) continue;//重儿子或者父节点，则跳过

dfs\_time(will, will);//新的一条链

}

}

上述代码中，dfs\_time为第二次DFS。主要思想是，如果有重儿子，那么就去遍历重儿子，这样可以保证一条重链的编号是连续的。

3、LCA部分代码

LCA(x,y)的主要是，找出原树中x,y这两个点之间所有的线段。这样，x到y的路径，被映射出若干个[L,R]的区间（所谓区间，这些数字是被重新标号了（dfs\_clock标的）！并不是原来的节点编号。）

int lca(int x, int y)//找出x到y的之间所有的区间

{

int ans = -inf;

//cout<<t[x].top <<" " << t[y].top<<endl;

while (t[x].top != t[y].top)

{

if (t[t[x].top].dep < t[t[y].top].dep) swap(x,y); //x深，y浅

//线段树查询begin

qans= -inf;//切记要初始化qans

ql=t[t[x].top].ti;

qr=t[x].ti;

query(1,1,n);

ans = max(ans, qans);

//线段树查询end

x = t[t[x].top].fa;

}

if (t[x].dep > t[y].dep) swap(x, y);//x是上面一些的节点

if (x != y)

{

//线段树查询begin

qans= -inf;

ql=t[x].ti + 1;//一定要+1，因为x,y在一条链上，所以标号连续。

qr=t[y].ti;

query(1,1,n);

ans = max(ans, qans);

//线段树查询end

}

return ans;

}

这段代码的主要思想是，如果2个节点不在一个链路中，那么让深度深的节点，直接跳到他的top节点，并且提取这段重链区间。

如果2个节点在一个链路重，那么就提取深度浅的节点，到深度深的节点的这段链路。

【根据题目意思不同，要考虑提取的是边，还是点。上述代码对应的提取的是边。对于点比较简单，对于提取边的主要思想是：给点标号以后，每个点的父边的标号与点的标号相同。 特判：1号点如果为根的话，那么1号点是没有父链的！】

### SPOJ的树剖代码

//----建边部分

struct Vector\_edge//存vector用的东西

{

int will, w;//v为连接的点， w为边权

Vector\_edge():will(0),w(0){}

Vector\_edge(int a, int b):will(a),w(b){}

};

vector<Vector\_edge>g[maxn];

inline void add\_edge(int a, int b, int c)

{

g[a].push\_back(Vector\_edge(b,c));

}

//----树链部分=====

int dfs\_clock;//DFS时间序列

struct node

{

int size, dep, son, top, fa, ti;

node():size(0),dep(0),son(0),top(0),fa(0),ti(0){};

//size:v为跟节点数量

//dep:v的深度 根为1

//top:v所在链的顶端

//son:vv在同一重链的儿子节点，重儿子

//ti:v与父亲的连边，在线段树中的位置

}t[maxn];

void dfs\_find(int u, int pa, int depth)//第一次DFS，得到size,dep,fa以及son的数据

{

t[u].son = 0; //重儿子，为0表示没有重儿子

t[u].size = 1; //节点数量

t[u].dep = depth;

t[u].fa = pa;

for (int i = 0; i != g[u].size(); ++ i)

{

int will = g[u][i].will;

if (will == pa) continue;

dfs\_find(will, u, depth + 1);

t[u].size += t[will].size;

if (t[will].size > t[t[u].son].size) t[u].son = will;

}

}

void dfs\_time(int u, int pa) // 得到时间戳等数据,u为当前节点，pa为父链顶端节点

{

t[u].ti = ++ dfs\_clock; //u这个节点的时间戳是dfs\_clock,dfs\_clock下标是从1开始的，没有0！

t[u].top = pa; //top是u所在链的顶端

if (t[u].son != 0) dfs\_time(t[u].son, t[u].top); //如果节点有重儿子，那么依旧是以pa为链顶端的一条链

for (int i = 0; i != g[u].size(); ++ i)

{

int will = g[u][i].will;

if (will == t[u].son || will == t[u].fa) continue;//重儿子或者父节点，则跳过

dfs\_time(will, will);//新的一条链

}

}

//----线段树部分--

int val[maxn<<2], mx[maxn<<2];

#define lson o\*2, L, M

#define rson o\*2+1, M + 1,R

int segment\_tree\_query\_l, segment\_tree\_query\_r, segment\_tree\_query\_ans;

//int segment\_tree\_update\_l, segment\_tree\_update\_r, segment\_tree\_update\_val;

//这种线段树写法，需要把询问的东西，放在全局变量

//qans就是query询问的结果，ql,qr是询问[L,R]区间的答案

//ql,qr,qans对于update的时候，本着不浪费的原则，复用一下，就不用upl,upr,和upval了

#define qans segment\_tree\_query\_ans

#define ql segment\_tree\_query\_l

#define qr segment\_tree\_query\_r

//upl ,upr是update的[L,R]区间，改为upval

#define upl segment\_tree\_update\_l

#define upr segment\_tree\_update\_r

#define upval segment\_tree\_update\_val

int n;

void update(int o, int L, int R)// L,R区间统一改为qans

{

if (ql <= L && R <= qr)

{

val[o] = mx[o] = qans;

return ;

}

int M = L + (R - L) / 2;

if (ql <= M) update(lson);

if (qr > M) update(rson);

mx[o] = max(mx[o\*2], mx[o\*2+1]);

}

void query(int o, int L, int R)

{

if (ql <= L && R <= qr)

{

qans= max(qans, mx[o]);

return ;

}

int M = L + (R- L)/2;

if (ql <= M) query(lson);

if (qr > M) query(rson);

}

int lca(int x, int y)//找出x到y的之间所有的区间

{

int ans = -inf;

//cout<<t[x].top <<" " << t[y].top<<endl;

while (t[x].top != t[y].top)

{

if (t[t[x].top].dep < t[t[y].top].dep) swap(x,y); //x深，y浅

//线段树查询begin

qans= -inf;//切记要初始化qans

ql=t[t[x].top].ti;

qr=t[x].ti;

query(1,2,n);

ans = max(ans, qans);

//线段树查询end

x = t[t[x].top].fa;

}

if (t[x].dep > t[y].dep) swap(x, y);//x是上面一些的节点

if (x != y)

{

//线段树查询begin

qans= -inf;

ql=t[x].ti + 1;//一定要+1，因为x,y在一条链上，所以标号连续。

qr=t[y].ti;

query(1,2,n);

ans = max(ans, qans);

//线段树查询end

}

return ans;

}

int da[maxn][3];//起点, 终点, 权重

void clear()

{

dfs\_clock = 0;

for (int i = 0; i != maxn; ++ i) g[i].clear();

memset(mx, 0, sizeof(mx));

memset(val, 0, sizeof(val));

}

int main()

{

int T; scanf("%d", &T);

while (T--)

{

clear();

scanf("%d", &n);

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

{

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

add\_edge(da[i][0], da[i][1], da[i][2]);

add\_edge(da[i][1], da[i][0], da[i][2]);

}

dfs\_find(1, 1, 1);

dfs\_time(1, 1);

for (int i = 1; i < n; i ++ )//人工更新所有的边

{

if (t[da[i][0]].dep > t[da[i][1]].dep)//相邻的边，一定是父子关系

swap(da[i][0] , da[i][1]);//da[i][1]是深度深的点, 然后da[i][1]这个点的DFS序号，就是他的前驱边的序号

//线段树update begin

ql = t[da[i][1]].ti;

qr = t[da[i][1]].ti;

qans = da[i][2];

update(1,2,n);

//线段树update end

}

char he[100];

while(1)

{

scanf("%s",he);

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

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

{

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

printf("%d\n",lca(a,b));

}

else

{

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

//线段树update begin

ql = t[da[a][1]].ti;

qr = t[da[a][1]].ti;

qans = b;

update(1,2,n);

//线段树update end

}

}

printf("\n");

}

return 0;

}

## 树状数组：

### 先上板子：

#### 一维：

1. const int maxn = 100;
2. #define lowbit(x) (x)&-(x)
3. struct BIT {//Binary Indexed Tree
4. int n; //下标从1开始
5. int C[maxn + 10];
6. void init(int n) { this->n = n + 5; memset(C, 0, sizeof(C)); }
7. int sum(int x, int ret = 0) { //x的前缀和（含x）
8. for (; x > 0; x -= lowbit(x))
9. ret += C[x];
10. return ret;
11. }
12. void add(int x, int d) { //点x 加 d
13. for (; x <= n; x += lowbit(x))
14. C[x] += d;
15. }
16. } g;

#### 二维：

1. const int maxn = 100;
2. #define lowbit(x) (x)&-(x)
3. struct BIT {//Binary Indexed Tree
4. int n, m; //下标从1开始
5. LL C[maxn + 10][maxn + 10];
6. void init(int n, int m) {
7. this->n = n + 5;
8. this->m = m + 5;
9. memset(C, 0, sizeof(C));
10. }
11. LL sum(int x, int y, LL ret = 0) { //x, y右下角的前缀和（含x y）
12. for (; x > 0; x -= lowbit(x))
13. for (int i = y; i > 0; i -= lowbit(i))
14. ret += C[x][i];
15. return ret;
16. }
17. void add(int x, int y, LL d) { //点x y 加 d
18. for (; x <= n; x += lowbit(x))
19. for (int i = y; i <= m; i += lowbit(i))
20. C[x][i] += d;
21. }
22. }g;

### 一维：

让树状数组完成区间加减，单点查询的功能。

直接做的话很困难，需要对问题做一些转化。

考虑将原数组差分，即令，特别地，。

此时，所以单点查询实际上就是在求d数组的[1..i]区间和。

而区间[l,r]整体加上的操作，可以简单地使用和来完成。

于是，我们用树状数组来维护d数组，就可以解决问题了。

下面再升级一次，完成区间加减，区间求和的功能。

仍然沿用d数组，考虑a数组[1,x]区间和的计算。d[1]被累加了x次，d[2]被累加了x-1次，...，d[x]被累加了1次。

因此得到



所以我们再用树状数组维护一个数组，即可完成任务。

POJ 3468就是这个功能的裸题，下面给出代码。

Sample Input

10 5

1 2 3 4 5 6 7 8 9 10

Q 4 4

Q 1 10

Q 2 4

C 3 6 3

Q 2 4

Sample Output

4

55

9

15

1. BIT g, f;
2. LL a[maxn];
3. int n, q;
4. void init()
5. {
6. scanf("%d%d", &n, &q);
7. for (int i = 1; i <= n; i++)
8. {
9. scanf("%lld", a + i);
10. }
11. f.init(n);
12. g.init(n);
13. }
14. void solve()
15. {
16. for (int i = 1; i <= n; i++)
17. {
18. g.add(i, a[i] - a[i-1]);
19. f.add(i, i \* (a[i] - a[i-1]));
20. }
21. char op[5];
22. LL L, R, add;
23. while (q--)
24. {
25. scanf("%s", op);
26. if (op[0] == 'Q')
27. {
28. scanf("%lld%lld", &L, &R); L--;
29. LL ansL = (L + 1) \* g.sum(L) - f.sum(L);
30. LL ansR = (R + 1) \* g.sum(R) - f.sum(R);
31. printf("%lld\n", ansR - ansL);
32. continue;
33. }
34. scanf("%lld%lld%lld", &L, &R, &add); R++;
35. g.add(L, add); g.add(R, -add);
36. f.add(L, L \* add); f.add(R, -R \* add);
37. }
38. }
39. int main(){
40. init();
41. solve();
42. return 0;

### 二维：

对于涉及矩形加减的情形，我们发现一维中的差分的办法在二维的情况用不出来，所以要改一下。思考一下一维中的差分的另外一个含义：d[i]同时也表示a[i..n]的整体增量，d[i]+=k就意味着把a[i]..a[n]全部加上了k。理解了之后就发现这个意义上可以推广到二维，仍假设原矩形初始全为0(不为0要可能要初始化，也可能有更好的方法，火车上想)，以便接下来的叙述。

令d[x][y]表示(x,y)-(n,m)矩形的整体增量，其中(n,m)是边界。

那么(x1,y1)-(x2,y2)矩形整体加k的代码就是





至此，矩形加减，单点查询的问题得到了解决。

重头戏在这里，矩形加减，矩形求和。

求原矩形(1,1)-(x,y)的和，结果由下式给出

sigma(i=1..x,j=1..y) a[i,j]\*(x-i+1)\*(y-j+1)

很好理解吧? 但是这个式子并不是那么容易求和的，展开一下求和的部分得到

a[i,j]\* ( (x+1)(y+1) - (x+1)\*j - (y+1)\*x + i\*j )

整个式子就是

(x+1)(y+1)sigma(a[i,j]) - (x+1)sigma(a[i,j]\*j) - (y+1)sigma(a[i,j]\*i) + sigma(a[i,j]\*i\*j)

知道怎么处理了吧？如果没有请回去复习一维的处理方法。

令b[i,j]=a[i,j]\*i c[i,j]=a[i,j]\*j d[i,j]=a[i,j]\*i\*j

维护a,b,c,d一共四个二维树状数组，问题得到解决。

tyvj p1716就是实现这两个功能的裸题，下面给出完整代码。

输入数据的第一行为X n m，代表矩阵大小为n×m。  
从输入数据的第二行开始到文件尾的每一行会出现以下两种操作：  
    L a b c d delta —— 代表将(a,b),(c,d)为顶点的矩形区域内的所有数字加上delta。  
    k a b c d     —— 代表求(a,b),(c,d)为顶点的矩形区域内所有数字的和。

1. BIT e, f, g, h;
2. int n, m;
3. void init()
4. {
5. char op[10];
6. scanf("%s", op);
7. scanf("%d%d", &n, &m);
8. e.init(n, m); //d[i][j]
9. f.init(n, m); //d[i][j] \* i
10. g.init(n, m); //d[i][j] \* j
11. h.init(n, m); //d[i][j] \* i \* j
12. }
13. void update(int x1, int y1, int x2, int y2, int delta)
14. {
15. x2++; y2++;
16. e.add(x1, y1, delta);
17. e.add(x2, y2, delta);
18. e.add(x1, y2, -delta);
19. e.add(x2, y1, -delta);
21. f.add(x1, y1, delta \* x1);
22. f.add(x2, y2, delta \* x2);
23. f.add(x1, y2, -delta \* x1);
24. f.add(x2, y1, -delta \* x2);
25. g.add(x1, y1, delta \* y1);
26. g.add(x2, y2, delta \* y2);
27. g.add(x1, y2, -delta \* y2);
28. g.add(x2, y1, -delta \* y1);
29. h.add(x1, y1, delta \* x1 \* y1);
30. h.add(x2, y2, delta \* x2 \* y2);
31. h.add(x1, y2, -delta \* x1 \* y2);
32. h.add(x2, y1, -delta \* x2 \* y1);
33. }
34. inline int sum(int x, int y)
35. {
36. return (x + 1) \* (y + 1) \* e.sum(x, y) - (y + 1) \* f.sum(x, y) - (x + 1) \* g.sum(x, y) + h.sum(x, y);
37. }
38. int get\_ans(int x1, int y1, int x2, int y2)
39. {
40. x1--; y1--;
41. return sum(x1, y1) + sum(x2, y2) - sum(x1, y2) - sum(x2, y1);
42. }
43. void solve()
44. {
45. char op[5];
46. int x1, x2, y1, y2, delta;
47. while (scanf("%s", op) == 1)
48. {
49. scanf("%d%d%d%d", &x1, &y1, &x2, &y2);
50. if (op[0] == 'L')
51. {
52. scanf("%d", &delta);
53. update(x1, y1, x2, y2, delta);
54. continue;
55. }
56. printf("%d\n", get\_ans(x1, y1, x2, y2));
57. }
58. }
59. int main()
60. {
61. init();
62. solve();
63. return 0;
64. }

## 网络流模型的一些建图

### 分数规划

分数规划的一般形式：

 其中，

式子中的可以是任何东西，点集，矩阵，数字，边集或者某些是否选取的01序列都可以。但是前提条件是。不可以等于0。

解决这样的问题，可以直接转化为求，其中，,为单调减函数，并且当的时候，为所求答案，同时的取值为最优解时的的取值。只要二分查找，然后用其他算法来求就可以解决问题了。

同样的，对于形如

 其中，

则直接转化为求，其中，,为单调减函数，并且当的时候，为所求答案，同时的取值为最优解时的的取值。

Hint：，有些算法求解的过程中，为了保证求得的值最小或者最大，可能会出现或者的情况。特别是用网络流求解最大密度子图时，把所有与源相连的边都割了。这种问题要十分小心。

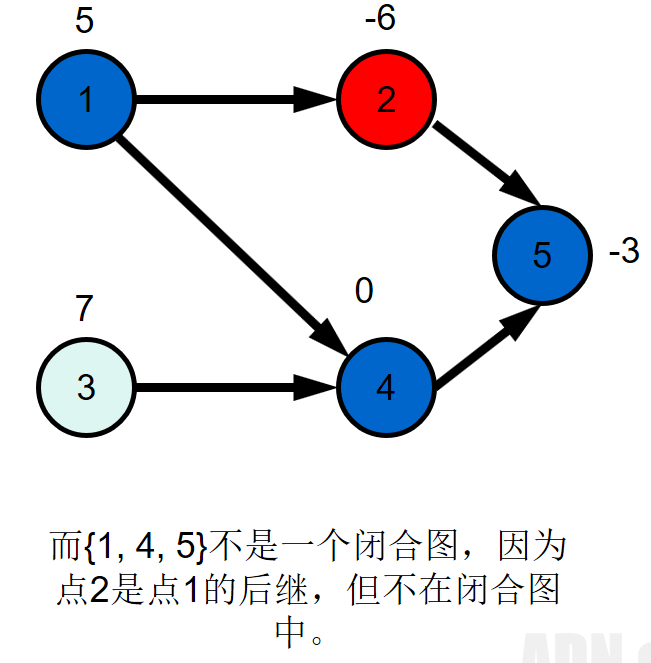
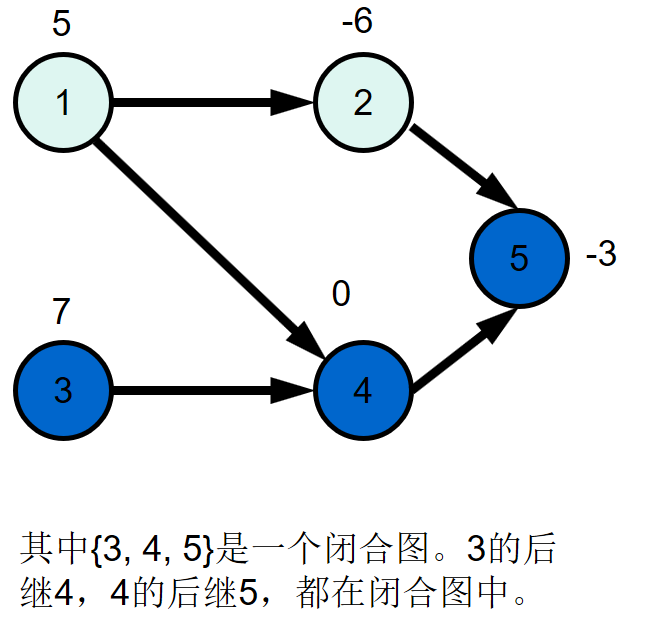
关于正确性的证明如下：对于，并且我们要求解的最小值。设。如果为最小值，此时的取值为，则，则，把右边式子分子乘到左边去，再移到右边得。设。我们来证为减函数。如果存在一个，那么当 ，取得最小值为。对于 而言，的话，,则显然因为，又因为式子的和相等，则的值肯定要比更小。说明了函数是单调减的。

对于当为最优解的充要性简单证明：假设时取得最优解，即，为非的其他取值。也就是说。简单的化简得。而可以化简为。这证明了，对于正确的，只有时的才是正确的取值(Dinkelbach定理)。正好，是单调减函数~可以方便计算。

假如存在一个更小的，是一个合法解（求依旧是）。正确解为。那么。把右边2个式子单独化简得，这和不符合。

### 最大权闭合图

所谓闭合图，就是有一个有向图，在这个有向图中的一个点集，该点所指向的所有点也在这个点集中。在布尔代数上，这叫做蕴含运算。显然，闭合图是允许超过一个连通块的。



最大权闭合图，自然就是点权和最大的闭合图。也就是”找一堆点，这些点出去的边所到达的点，依然在这堆点当中。找出权和最大的点集。”

构图方法：



然后直接从源到汇跑一次最大流。最小割，割开了源和汇所在的集合。而所在的集合，除去以外的点，就是最大权闭合图中，应该被选中的点。

同时，如果所有点的权重为，最大流或者最小割(两者相等)为,那么最大权闭合图能取得的总权重为。从出发，如果有一条边到，假设边的流量小于容量（没满流），则视为连通。从出发，遍历所有能遍历到的点，除去外的点，就是选中的最大权闭合图。

### 求解Maximize|Ei|-|Vi|



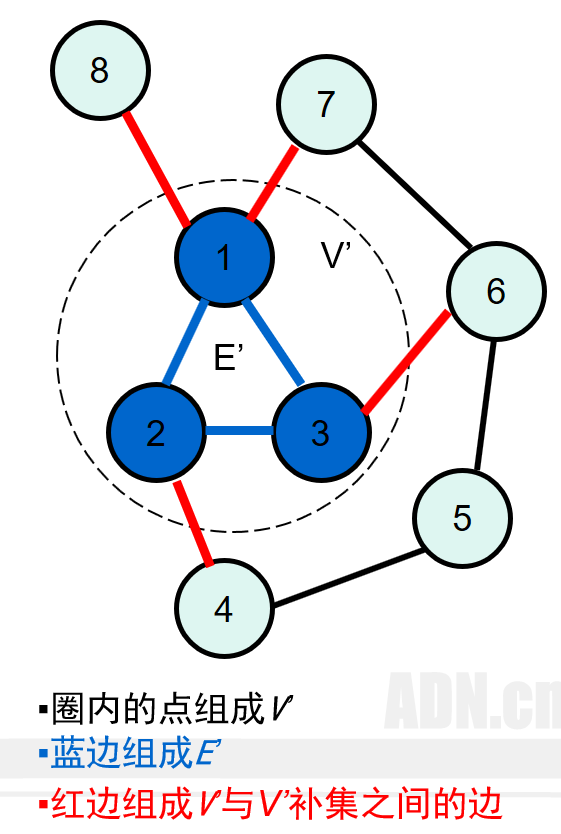
标题的式子不完整（为了看起来方便）。为从一张图中，选出的一些点（是一个集合）。为选出的点的导出子图（导出子图就是，选出的点之间的边保留，其他边都扔掉）的边（因为有多个边，也是一个集合）。点，边有权，求一个导出子图，边权和减去点权和的最大值。

假设， 为边的权重。

如果把题目视为，先选出一些顶点，为（是一个集合），显然就是没选的点了。然后两个集合之间的边，就很像一个割集。假设选一个点，这个点对答案的贡献是：将会付出这个点权的代价，但是会得到所有与这个点相连的边的代价，减去这个点与集合点相连边的代价。但是得到的代价这部分，会被算两次（因为一条边有2个端点，每个端点都计算了一次这个边的权重。），所以还要除以2。

既然是求最大值，那么我们把问题扩大为，对于最优解而言，点集的取值方案一定不变。所以，我们把上述每个点对答案的贡献乘以2，那么每个点对答案的贡献就是：

1. 付出(减去)这个点权重的二倍
2. 得到(加上)与这个点相连的所有边的权重，减去这个点与集合点相连边的边权。（补集转换思想）



对于所有被选中的点，他们的“与这个点与集合点相连边的代价”总和，就是一个割。所以我们可以整合所有的点，也就是上文说的“减去与这个点与集合点相连边的代价”这个贡献。所有选中的点的贡献之和，就是一个割。也就是上图中的红色的边。

建图方法：首先假设一个足够大的数字，来确保每条边流量非负。通常，的取值为, 为每个边的权重。也就是说，**取所有边的权重之和。

假设， 为边的权重。

1、新建新的源和新的汇。

2、对于原图中的所有边，改为正反两条边，流量为原来的边权。

3、对于原图中所有点，源连边到每个点，流量为。

4、对于原图中所有点，都连向汇，流量为,其中为点权，。也就是为与这个点相连的所有的边的权重之和。

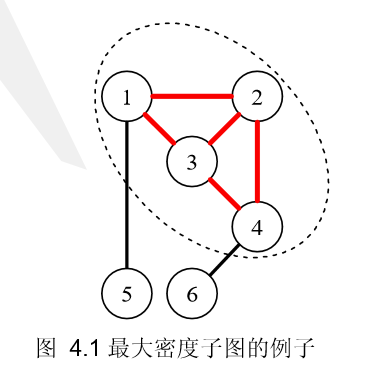
然后直接从源到汇跑一次最大流。最小割，割开了源和汇所在的集合。而所在的集合，除去以外的点，就是图中，应该被选中的点。

并且，的最优解就是。其中就是到的最小割，为原图中点的总量（对于网络流的图而言，比原图多了源与汇两个点）。

，

### 最大密度子图

最大密度子图，就是选出一些点，这些点之间的所有的边的数量为，选出的点的数量为，使得最大。



上面的是形如分数规划的式子，重写式子，得。其中为选出的点的导出子图的边集，边的数量(上图中，选出的点集显然为1,2,3,4这个4个点。导出子图的边集就是红色的边)；显然就是选中的点（上图中点标号为1,2,3,4的点）的数量（上图为4个）。这显然是一个分数规划问题。

直接可以根据分数规划的套路，改写为关于的函数, 去二分。剩下的问题就是求的问题了。这一看就是的问题。边权为1，所有点权都为。然后用的套路解决即可。对于最大密度子图有一个套路，二分的精度，只要（其中是原图中点的总数，和为二分的上下边界），那么一般就一定满足精度要求了。

1. 无源汇，有下界的可行流（部分边要求经过流量有下界）

可行流，并不要求流量是最大！而且原图为循环流（所有点入度都不为0）

首先转化为有源汇的网络流。新增源与汇。

构图方法：原图中对于到的边而言，为到的下界流量。为到的容量（上界流量），为到经过的实际流量。对于点,从出去的下界流量为,那么进入的下界流量为。

0、增源与汇

1、对于原来每一条边，方向不变，流量改为

2、对于每一个点，如果，则从到一条边，流量为。

3、对于每一个点，如果，则从到一条边，流量为。

与从源到汇跑一次最大流，如果与、相连的所有边都满流，则存在一个可行流。

### 有源汇，有下界的可行流

可行流，并不是最大流！

汇连向源，流量为无穷大，下界为0。

不用新建源，汇，直接按照无源汇的构图法构图和求解即可。

### 有源汇，有下界最大流

这次是最大流，不是可行流

方法1：按照有源汇，可行流的方法构图，但是，对于汇连向源，流量为无穷大的边，下界不再是0，而是，二分这个。如果,会得到可行流无解，否则只要可行流有解，就满足。

方法2：从汇向源连一条上界为无穷，下界为0的边，然后：

1. 按照无源汇的方法，新建附加源与附加汇变为无源汇。
2. 进行无源汇相关构图
3. 跑一次最大流
4. 拆掉原图中的附加边
5. 求的最大流。

### 有源汇，上下界最小流

方法1：同有源汇最大流的二分法，只不过原图新建的边的上界为，下界为0,二分这个，如果，那么存在可行流，否则不存在可行流。

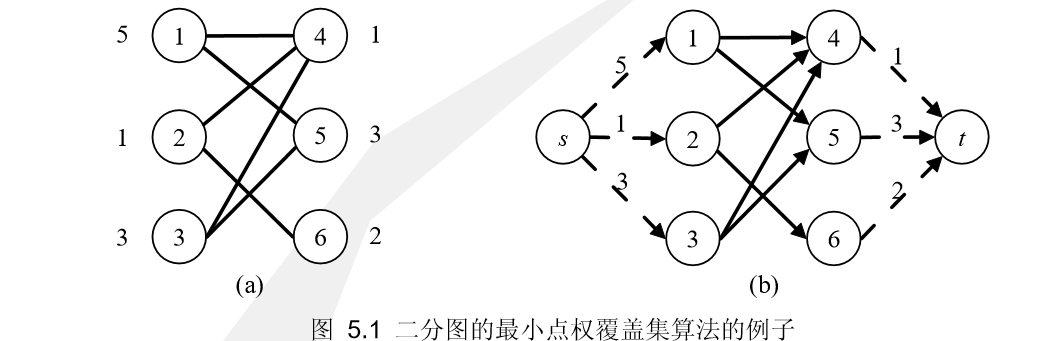
方法2：

1. 不添加！添加附加源汇，直接按照无源汇可行流法构边。
2. 跑最大流
3. 添加的边，流量无穷，下界为0
4. 跑最大流
5. 如果出去的边，全部满流，边的流量即为答案。如果出去的边流量没全满，则无解

### 在二分图中，用最少的点权，覆盖所有的边(二分图的最小点权覆盖集)。

构图方法：

1. 新建附加源与附加汇。
2. 对于二分图中左边每个点，都连一条边。流量为的权重
3. 二分图右边每个点，连一条边，流量为位的权重。
4. 原图中的的边（双向，单向），只保留的边，流量为无穷，也就是只保留了从图左边到图右边的边。



5、跑的最大流。所有在最后没有与相连的点（边的流量满了），都是被选中的点。

### 二分图的最大点权独立集算法

用最大的点权，所有点，两两不相连。

方法：求出二分图的最小点权覆盖集，这个覆盖集的的补集，就是二分图的最大点权独立集。也就是说，先用求出用最少的点权覆盖所有的边的点，剩下的点就是最大点权彼此不相连的点。

### 费用与流量平方成正比

对于一条边，容量为，费用为（如果经过流量为，那么费用为）。只需要把边拆开，拆成条边，每条边的费用分别为,,,……然后直接跑正常的费用流即可。

### 刘汝佳书P367的流量不固定最小费用流

### 公平分配问题

有n个任务，m处理器。每个人任务都需要分配给一个处理器完成，但是每个任务并不是所有处理器都能处理的，必须是指定的几个处理器当中的任意一个。求一个分配方案，使得最终处理任务最多的处理器，处理的任务最少。

构图方法：

1. 新建源与汇
2. 连每一个任务，流量为1
3. 任务如果能被处理器完成，则连边，流量为1
4. 每个任务，与连边，流量为
5. 二分，从跑最大流，如果与相连的边都满流，则。

### 区间K覆盖问题

数轴上有一些权值左开右闭的区间，找出权和最大的一些区间，使得任意一个数字，最多被k个区间覆盖。

构图：

1. 权和为的区间，加边，容量为1，费用为
2. 任何相邻的点加边,容量为,费用为0.
3. 求最左边点到最右边点的最小费用最大流。每一个流对应一组互不相交的区间。
4. 数据范围大考虑离散化

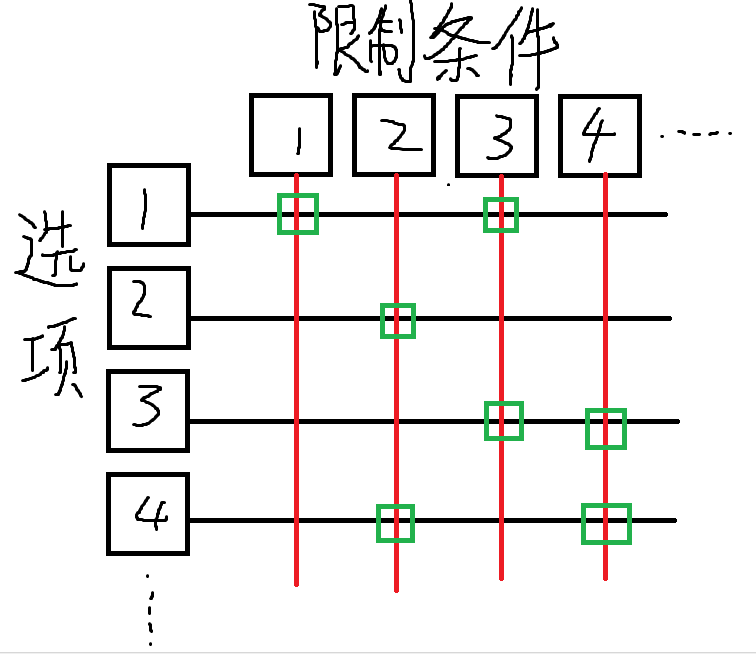
## 舞蹈链Dancing links

### 简介

DLX主要解决的是精确覆盖问题。这个问题还分为可重复与不可重复覆盖两种。所谓可重复覆盖，大概就是每个点可以被覆盖多次。不可重复的覆盖，就是每个点只能被覆盖一次。

经典的不可重复覆盖主要就是数独问题。可重复覆盖问题，后面也会用一个例子来进行简单的讲解。

主要格式（思想）如下：



横着的一排为限制条件，通常为需要满足的东西。比如每个格子是否填上数字，每个怪兽是否被消灭……

竖着一排为选项，用来满足限制条件的。比如覆盖若干个位置的限制条件。

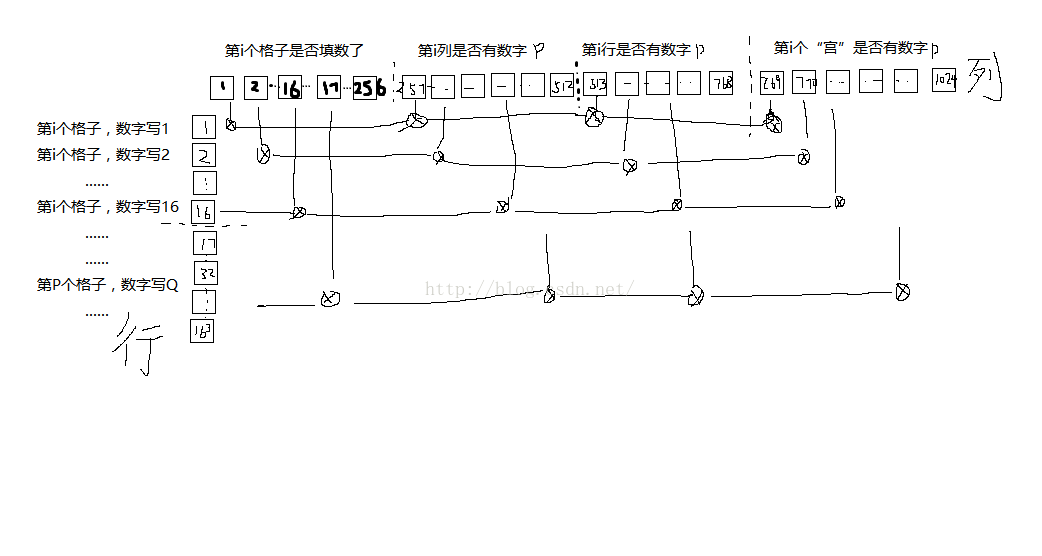
上图中，有4个条件需要被满足。有4个选项，选项1可以覆盖1,3的条件。选项2可以覆盖条件2。 选项3可以覆盖3,4。 选项4可以覆盖2,4。可重复覆盖的话，就是一个限制条件可以被多个选项覆盖，不可覆盖的话则一个限制条件只能被选项覆盖一次。

### 不可重复覆盖代码板子（数独板子）来自kuangbin的板子

1. const int N = 16; //4\*4数独
2. const int MaxN = N\*N\*N + 10;
3. const int MaxM = N\*N\*4 + 10;
4. const int maxnode = MaxN\*4 + MaxM + 100;
5. struct DLX //感谢kuangbin的模板
6. {
7. int n,m,size;
8. int U[maxnode],D[maxnode],R[maxnode],L[maxnode],Row[maxnode],Col[maxnode];
9. int H[MaxN],S[MaxM];
10. int ansd,ans[MaxN];
11. void init(int \_n,int \_m)//n行，m列的，DLX中操纵的元素，都是以1为开始下标
12. {
13. n = \_n;
14. m = \_m;
15. for(int i = 0;i <= m;i++)
16. {
17. S[i] = 0;
18. U[i] = D[i] = i;
19. L[i] = i-1;
20. R[i] = i+1;
21. }
22. R[m] = 0; L[0] = m;
23. size = m;//DLX中的链表节点总数
24. for(int i = 1;i <= n;i++)H[i] = -1;
25. }
26. void Link(int r,int c)
27. {
28. ++S[Col[++size]=c];
29. Row[size] = r;
30. D[size] = D[c];
31. U[D[c]] = size;
32. U[size] = c;
33. D[c] = size;
34. if(H[r] < 0)H[r] = L[size] = R[size] = size;
35. else
36. {
37. R[size] = R[H[r]];
38. L[R[H[r]]] = size;
39. L[size] = H[r];
40. R[H[r]] = size;
41. }
42. }
43. void remove(int c)
44. {
45. L[R[c]] = L[c]; R[L[c]] = R[c];
46. for(int i = D[c];i != c;i = D[i])
47. for(int j = R[i];j != i;j = R[j])
48. {
49. U[D[j]] = U[j];
50. D[U[j]] = D[j];
51. --S[Col[j]];
52. }
53. }
54. void resume(int c)
55. {
56. for(int i = U[c];i != c;i = U[i])
57. for(int j = L[i];j != i;j = L[j])
58. ++S[Col[U[D[j]]=D[U[j]]=j]];
59. L[R[c]] = R[L[c]] = c;
60. }
61. bool Dance(int d)
62. {
63. if(R[0] == 0)
64. {
65. for(int i = 0;i < d;i++)
66. {
67. output[(ans[i]-1)/16] = (char)((ans[i]-1)%16 + 'A');
68. }
69. int t=0;
70. for (int i = 0; i <N ;++i)
71. {
72. char ot[20];
73. for (int j = 0; j < N; ++ j)
74. ot[j] = output[t++];
75. ot[17] =0;
76. printf("%s\n", ot);
78. }
79. return true;
80. }
81. int c = R[0];
82. for(int i = R[0];i != 0;i = R[i])
83. if(S[i] < S[c])
84. c = i;
85. remove(c);
86. for(int i = D[c];i != c;i = D[i])
87. {
88. ans[d] = Row[i];
89. for(int j = R[i];j != i;j = R[j])remove(Col[j]);
90. if(Dance(d+1))return true;
91. for(int j = L[i];j != i;j = L[j])resume(Col[j]);
92. }
93. resume(c);
94. return false;
95. }
96. }dlx;

#### 数独的讲解，以及上述板子的介绍

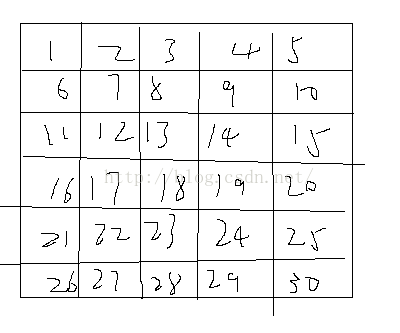
1. 数独讲解



上图就是数独的DLX的构图方法。常见的数独差不多就是要求每行、每列、每宫都要有特定的数字，并且每个格子都得填上数。

常见的数独有一些特殊的小技巧。对于N\*N宫的数独（3\*3宫，就是9\*9的数独）

（1）如果给每个格子编号的话，是这样的形式来编号的（如下图所示）



（2）因为DLX有一行限制条件与一列选项，所以我们需要根据数独的特点，来编号。假设数独棋盘的坐标为(r,c)的位置写上t。（r为竖着的坐标，c为横着的坐标，也就是常见的计算机的i,j坐标系）

可以得到如下结论：（N为N\*N宫的数独表）

所在的格子数字编号为：sn = (r-1) \* N \* N + c

(r,c,t)所在的行号为：(sn-1) \* N \*N + t;

(r,c,t)所在的【是否填数列，第i格子是否有数】列号:sn

(r,c,t)所在【第r行是否填了t】列号：N\*N \* N\*N+(c-1) \* (N\*N) + t

(r,c,t)所在【第c列是否填了t】列号：2 \* N \* N \* N\*N + (r-1) \* (N \* N) + t

(r,c)所属宫号为：p =  N\* [ ceil(1.0 \* r / N) - 1] + ceil(1.0 \* c / N)

(r,c,t)所在【第p宫是否填了t】的列号为： 3 \* N \* N \* N\*N +(p - 1) \* (N \* N) + t

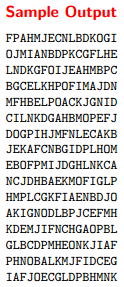
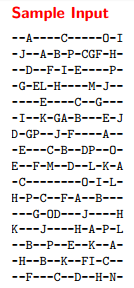
常见的代码为：

1. void place(int &q, int &a, int &b, int &c, int &d, int row, int col, int t)
2. //q为返回的行号(从1开始)，a,b,c,d依次为给出的4列。 row为传入行号，下标从1开始。col为列号，下标从1开始。t为数独填入的数字（从1开始）
3. {
4. int N = 4;//4\*4宫
5. int sn = (row-1) \* N \* N + col;
6. int p = N \* ( ceil(1.0 \* row / N) - 1) + ceil(1.0 \* col / N);
7. q = (sn-1) \* N \*N + t;
8. a = sn;
9. b = N \* N \* N \* N +(col-1) \* (N\*N) + t;
10. c = 2 \* N \* N \* N \* N + (row-1) \* (N \* N) + t;
11. d = 3 \* N \* N \* N \* N +(p - 1) \* (N \* N) + t;
12. }

用这一段代码，来计算数独的DLX的各个“交叉节点”。

例题1：

题目大意：给定一个数独，左边为给定的，右边为答案。保证答案唯一。



输出一个答案。ABC……分别对应1,2,3…… 。 这题是一个4\*4宫的数独，也就是16\*16的数独。

我们按照上述的方案，对于-符号，也就是可以任意填写的，那么我们就穷举一遍A~P的所有选项加入DLX中，对于其他已经确定的方案，就把确定的方案放入DLX中。因为每个格子只会被构造一次，所以这样可以保证正确性（请略加思考）。因为题目保证答案唯一，那么直接输出答案即可。也就是在Dance函数中，一旦满足R[0] == 0，那么程序就输出相关数据，程序结束。

例1代码在附录。

1. 板子介绍

dlx.init(X,Y); 这个函数，表示初始化一个DLX。这个DLX有X行，Y列。

Link(r,c)表示在DLX的r,c节点添加一个“机关”。也就是本文第一张图绿色的部分。用来表示c列可以被r行覆盖。

Dance(d)也就是核心函数了，用来运行DLC开始进行搜索。一旦满足R[0] == 0，说明搜到了一个可行解，并且这里会返回一个true。搜不到解则返回false。

Dance(d)函数有一段代码如下

1. remove(c);
2. for(int i = D[c];i != c;i = D[i])
3. {
4. ans[d] = Row[i]; //第d列，被第Row[i]行覆盖。
5. for(int j = R[i];j != i;j = R[j])remove(Col[j]);
6. if(Dance(d+1))return true; //如果已经搜到答案了，直接return true
7. for(int j = L[i];j != i;j = L[j])resume(Col[j]);
8. }

resume(c); 其中ans[d] = Row[i]; 表示第d列，被第Row[i]行覆盖。Row[i]就知道i是第几行了。

主程序中直接调用dlx.Dance(0); 就开始“跳舞”了

1. 可重复覆盖代码板子

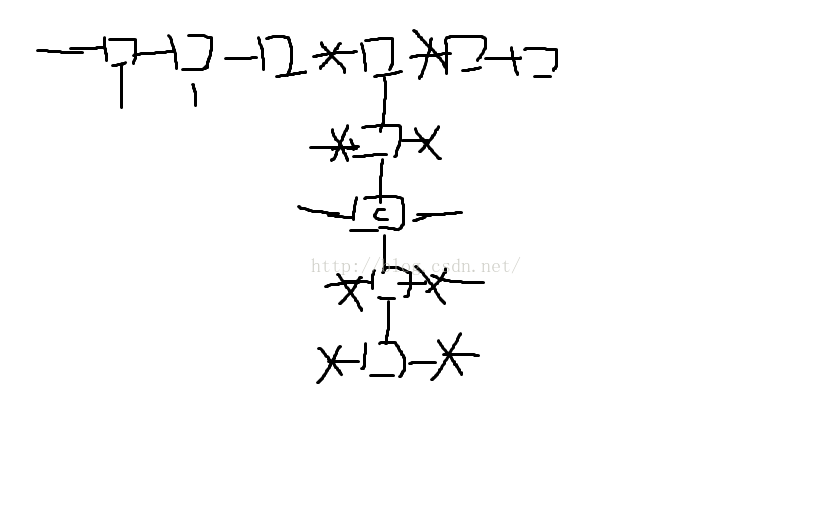
所谓可重复覆盖，就是每个限制条件可以被多个选项覆盖。出于某些原因，会要求求出最大最小值之类的。

所以个别操作略有区别。

1、remove

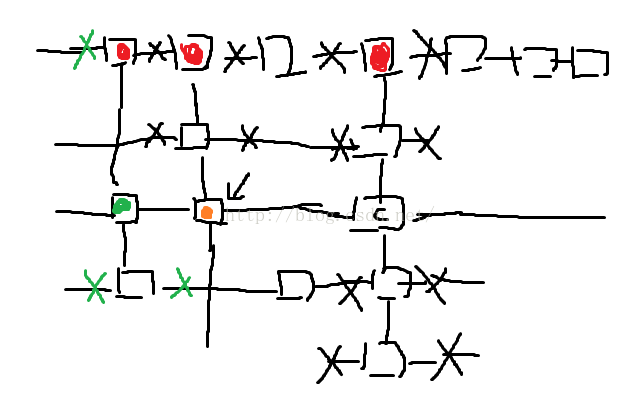
1. void remove(int c) //从c开始，往下，把所有元素都和左右隔离开
2. {
3. for(int i = D[c];i != c;i = D[i])
4. L[R[i]] = L[i], R[L[i]] = R[i];
5. }

remeve操作对于要删c元素，把除了c元素以外所有的这一列上的元素，都和所在行分割开。如下图所示



这样，如果可以覆盖的话，第一行的标记一定切割了。（就是图中，c上面顶端的那个，也被左右方向隔开了）

2、一次完整的删除操作



如上图，对于选中的C元素的时候，会先删c这一列，再删黄色节点，最后删绿色节点。删完以后，再去dance(d+1)，这个时候，我们虽然删了第二行，但是依旧可以选择第三行（重复覆盖了c所在列）。从图中可以看出，网格变稀疏了。

3、板子代码

1. const int MaxM = 15\*15+10;
2. const int MaxN = 15\*15+10;
3. const int maxnode = MaxN \* MaxM;
4. const int INF = 0x3f3f3f3f;
5. struct DLX
6. {
7. /\*
8. \* 可重复覆盖，每次为删除一行
9. \*/
10. int n,m,size;
11. int U[maxnode],D[maxnode],R[maxnode],L[maxnode],Row[maxnode],Col[maxnode];//maxnode表示总的节点数量
12. int H[MaxN],S[MaxM];
13. int ansd;
14. void init(int \_n,int \_m) //初始化为n行，m列
15. {
16. n = \_n;
17. m = \_m;
18. for(int i = 0;i <= m;i++)
19. {
20. S[i] = 0; //i列元素数量,这里初始化为0
21. U[i] = D[i] = i;
22. L[i] = i-1;
23. R[i] = i+1;
24. }
25. R[m] = 0; L[0] = m;
26. size = m; //总元素数量，用来做数组用
27. for(int i = 1;i <= n;i++)H[i] = -1; //每一行的H为空
28. ansd = INF;
29. }
30. void Link(int r,int c) //r行,c列有元素，添加这个元素
31. {
32. ++S[Col[++size]=c];//S[c]++表示c列元素数量增加， col[k]表示第k个元素是第几列
33. Row[size] = r; //row[k]第k个元素是第r行
34. D[size] = D[c];
35. U[D[c]] = size;
36. U[size] = c;
37. D[c] = size; //链表的互相操作
38. if(H[r] < 0)H[r] = L[size] = R[size] = size; //更改行的相关指针
39. else
40. {
41. R[size] = R[H[r]];
42. L[R[H[r]]] = size;
43. L[size] = H[r];
44. R[H[r]] = size;
45. }
46. }
47. void remove(int c) //从c开始，往下，把所有元素都和左右隔离开
48. {
49. for(int i = D[c];i != c;i = D[i])
50. L[R[i]] = L[i], R[L[i]] = R[i];
51. }
52. void resume(int c) //恢复第c列
53. {
54. for(int i = U[c];i != c;i = U[i])
55. L[R[i]] = R[L[i]] = i;
56. }
57. bool v[MaxM];
58. int f() //剪枝，至少还要选择多少次，才能全部选择完
59. {
60. /\*思想：对于一个没有被选择的列，如果选择这一列的时候，能把所有和这一列相关的列都选中，\*/
61. int ret = 0;
62. for(int c = R[0]; c != 0;c = R[c])v[c] = true;//表示c列还没有删
63. for(int c = R[0]; c != 0;c = R[c])
64. if(v[c]) //如果c列没有被删
65. {
66. ret++; //至少要选一个,但是我们把所有的都给选上了，这一定是最少的方案
67. v[c] = false;
68. for(int i = D[c];i != c;i = D[i])//[1...m]都是初始点！，所以一开始从D[c]开始
69. for(int j = R[i];j != i;j = R[j])
70. v[Col[j]] = false;
71. }
72. return ret;
73. }
74. void Dance(int d)
75. {
76. //if(d + f() >= ansd)return;
77. if(R[0] == 0)
78. {
79. if(d < ansd)ansd = d;
80. return;
81. }
82. int c = R[0];
83. for(int i = R[0];i != 0;i = R[i])//找到第c列，还没有被消除的
84. if(S[i] < S[c])
85. c = i;
86. for(int i = D[c];i != c;i = D[i]) //穷举消除
87. {
88. remove(i);//第c列第i行
89. for(int j = R[i];j != i;j = R[j])remove(j); //删除这一行,因为是可重复的，所以值要删这一行的所有元素，同时这一行元素的列也删去
90. Dance(d+1);
91. for(int j = L[i];j != i;j = L[j])resume(j);//恢复
92. resume(i);
93. }
94. }
95. }dlx;

从上述板子可以看出，其实和不可重复覆盖区别并不大。Dance(d)函数第一行

if(d + f() >= ansd)return;

这是用来剪枝的，一些题目要求用最少的选项来覆盖所有的太偶见，其中

d是已经选择的选项，

f函数则为估价。估算未来最少用多少个选项就可以覆盖所有的限制条件。

ansd为已经得到的最优答案，如果搜下去不可能更优，显然直接return即可。

1. 剪枝（求最少的选项覆盖所有的限制条件）

对于求最少删除方案，我们可以最一个最好的估价。

假设我们要覆盖第i列，我们一次选择所有能覆盖第i列的行（因为我们只有这么多选项），这样就又有很多列被覆盖了。找到下一个没有被覆盖的列，还是一次选择所有能覆盖这一列的行，然后继续下去。

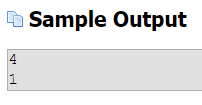
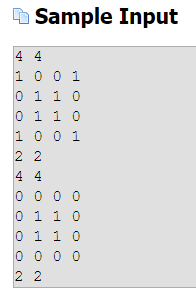
虽然一次选择多行，但是我们记为一次选择。 这样的总选择数，就是最好的结果。

然后用当前步数+估价，和当前得到最优答案进行比较，判断剪枝即可

1. 例题讲解

例2：FOJ 1686

有一个n\*m的网格，某些格子有怪兽。0表示无，1表示有。有一个人，每次可以让一个x\*y的格子的怪兽全部死。问，这个人最少多少次可以让所有怪兽全部死亡。



上图为样例，2组数据。第一组数据是4\*4的网格，那个人可以攻击2\*2的范围。需要放4次技能才能杀掉所有的怪兽。

解法：

套用上述的可重复覆盖的DLX。每个怪兽编号，每个技能可以放在地图上每一个地方，给技能编号。DLX的时候剪一下枝即可。代码见附录

### 附录

1. 例1的AC代码。
   1. char output[17\*17];
   2. const int N = 16; //4\*4数独
   3. const int MaxN = N\*N\*N + 10;
   4. const int MaxM = N\*N\*4 + 10;
   5. const int maxnode = MaxN\*4 + MaxM + 100;
   6. struct DLX //感谢kuangbin的模板
   7. {
   8. int n,m,size;
   9. int U[maxnode],D[maxnode],R[maxnode],L[maxnode],Row[maxnode],Col[maxnode];
   10. int H[MaxN],S[MaxM];
   11. int ansd,ans[MaxN];
   12. void init(int \_n,int \_m)//n行，m列的，DLX中操纵的元素，都是以1为开始下标
   13. {
   14. n = \_n;
   15. m = \_m;
   16. for(int i = 0;i <= m;i++)
   17. {
   18. S[i] = 0;
   19. U[i] = D[i] = i;
   20. L[i] = i-1;
   21. R[i] = i+1;
   22. }
   23. R[m] = 0; L[0] = m;
   24. size = m;//DLX中的链表节点总数
   25. for(int i = 1;i <= n;i++)H[i] = -1;
   26. }
   27. void Link(int r,int c)
   28. {
   29. ++S[Col[++size]=c];
   30. Row[size] = r;
   31. D[size] = D[c];
   32. U[D[c]] = size;
   33. U[size] = c;
   34. D[c] = size;
   35. if(H[r] < 0)H[r] = L[size] = R[size] = size;
   36. else
   37. {
   38. R[size] = R[H[r]];
   39. L[R[H[r]]] = size;
   40. L[size] = H[r];
   41. R[H[r]] = size;
   42. }
   43. }
   44. void remove(int c)
   45. {
   46. L[R[c]] = L[c]; R[L[c]] = R[c];
   47. for(int i = D[c];i != c;i = D[i])
   48. for(int j = R[i];j != i;j = R[j])
   49. {
   50. U[D[j]] = U[j];
   51. D[U[j]] = D[j];
   52. --S[Col[j]];
   53. }
   54. }
   55. void resume(int c)
   56. {
   57. for(int i = U[c];i != c;i = U[i])
   58. for(int j = L[i];j != i;j = L[j])
   59. ++S[Col[U[D[j]]=D[U[j]]=j]];
   60. L[R[c]] = R[L[c]] = c;
   61. }
   62. bool Dance(int d)
   63. {
   64. if(R[0] == 0)
   65. {
   66. //搜出一个解了,开始计算并输出相关东西
   67. for(int i = 0;i < d;i++) //穷举每一个格子，看看每一个格子被哪一列覆盖了
   68. {
   69. output[(ans[i]-1)/16] = (char)((ans[i]-1)%16 + 'A');
   70. }
   71. int t=0;
   72. for (int i = 0; i <N ;++i)
   73. {
   74. char ot[20];
   75. for (int j = 0; j < N; ++ j)
   76. ot[j] = output[t++];
   77. ot[17] =0;
   78. printf("%s\n", ot);
   79. }
   80. return true;
   81. }
   82. int c = R[0];
   83. for(int i = R[0];i != 0;i = R[i])
   84. if(S[i] < S[c])
   85. c = i;
   86. remove(c);
   87. for(int i = D[c];i != c;i = D[i])
   88. {
   89. ans[d] = Row[i]; //第d列，被第Row[i]行覆盖。
   90. for(int j = R[i];j != i;j = R[j])remove(Col[j]);
   91. if(Dance(d+1))return true; //如果已经搜到答案了，直接return true
   92. for(int j = L[i];j != i;j = L[j])resume(Col[j]);
   93. }
   94. resume(c);
   95. return false;
   96. }
   97. }dlx;
   98. void place(int &q, int &a, int &b, int &c, int &d, int row, int col, int t)
   99. //q为返回的行号(从1开始)，a,b,c,d依次为给出的4列。 row为传入行号，下标从1开始。col为列号，下标从1开始。t为数独填入的数字（从1开始）
   100. {
   101. int N = 4;
   102. int sn = (row-1) \* N \* N + col;
   103. int p = N \* ( ceil(1.0 \* row / N) - 1) + ceil(1.0 \* col / N);
   104. q = (sn-1) \* N \*N + t;
   105. a = sn;
   106. b = N \* N \* N \* N +(col-1) \* (N\*N) + t;
   107. c = 2 \* N \* N \* N \* N + (row-1) \* (N \* N) + t;
   108. d = 3 \* N \* N \* N \* N +(p - 1) \* (N \* N) + t;
   109. }
   110. char s[200];
   111. void makeit(int row)
   112. {
   113. for (int i = 1; i <= 16; ++ i)
   114. {
   115. int q, a,b,c,d;
   116. if (s[i] != '-')
   117. {
   118. place(q,a,b,c,d,row,i,s[i]-'A'+1);
   119. dlx.Link(q, a);
   120. dlx.Link(q, b);
   121. dlx.Link(q, c);
   122. dlx.Link(q, d);
   123. continue;
   124. }
   125. for (int t = 1; t <= 16; ++ t) //穷举16个选择方法
   126. {
   127. place(q,a,b,c,d,row,i,t);
   128. dlx.Link(q, a);
   129. dlx.Link(q, b);
   130. dlx.Link(q, c);
   131. dlx.Link(q, d);
   132. }
   133. }
   134. }
   135. void init()
   136. {
   137. dlx.init(N\*N\*N,N\*N\*4);
   138. makeit(1);
   139. for (int i = 2; i <= 16;++i)
   140. {
   141. scanf("%s", s + 1);
   142. makeit(i);
   143. }
   144. }
   145. int main()
   146. {
   147. int sb=0;
   148. while (scanf("%s", s + 1) != EOF)
   149. {
   150. if (sb) printf("\n");
   151. ++sb;
   152. init();
   153. dlx.Dance(0);
   154. }
   155. return 0;
   156. }

二、例2的AC代码。

1. int n, m;
2. const int MaxM = 15\*15+10;
3. const int MaxN = 15\*15+10;
4. const int maxnode = MaxN \* MaxM;
5. const int INF = 0x3f3f3f3f;
6. struct DLX
7. {
8. /\*
9. \* 可重复覆盖，每次为删除一行
10. \*/
11. int n,m,size;
12. int U[maxnode],D[maxnode],R[maxnode],L[maxnode],Row[maxnode],Col[maxnode];//maxnode表示总的节点数量
13. int H[MaxN],S[MaxM];
14. int ansd;
15. void init(int \_n,int \_m) //初始化为n行，m列
16. {
17. n = \_n;
18. m = \_m;
19. for(int i = 0;i <= m;i++)
20. {
21. S[i] = 0; //i列元素数量,这里初始化为0
22. U[i] = D[i] = i;
23. L[i] = i-1;
24. R[i] = i+1;
25. }
26. R[m] = 0; L[0] = m;
27. size = m; //总元素数量，用来做数组用
28. for(int i = 1;i <= n;i++)H[i] = -1; //每一行的H为空
29. ansd = INF;
30. }
31. void Link(int r,int c) //r行,c列有元素，添加这个元素
32. {
33. ++S[Col[++size]=c];//S[c]++表示c列元素数量增加， col[k]表示第k个元素是第几列
34. Row[size] = r; //row[k]第k个元素是第r行
35. D[size] = D[c];
36. U[D[c]] = size;
37. U[size] = c;
38. D[c] = size; //链表的互相操作
39. if(H[r] < 0)H[r] = L[size] = R[size] = size; //更改行的相关指针
40. else
41. {
42. R[size] = R[H[r]];
43. L[R[H[r]]] = size;
44. L[size] = H[r];
45. R[H[r]] = size;
46. }
47. }
48. void remove(int c) //从c开始，往下，把所有元素都和左右隔离开
49. {
50. for(int i = D[c];i != c;i = D[i])
51. L[R[i]] = L[i], R[L[i]] = R[i];
52. }
53. void resume(int c) //恢复第c列
54. {
55. for(int i = U[c];i != c;i = U[i])
56. L[R[i]] = R[L[i]] = i;
57. }
58. bool v[MaxM];
59. int f() //剪枝，至少还要选择多少次，才能全部选择完
60. {
61. /\*思想：对于一个没有被选择的列，如果选择这一列的时候，能把所有和这一列相关的列都选中，\*/
62. int ret = 0;
63. for(int c = R[0]; c != 0;c = R[c])v[c] = true;//表示c列还没有删
64. for(int c = R[0]; c != 0;c = R[c])
65. if(v[c]) //如果c列没有被删
66. {
67. ret++; //至少要选一个,但是我们把所有的都给选上了，这一定是最少的方案
68. v[c] = false;
69. for(int i = D[c];i != c;i = D[i])//[1...m]都是初始点！，所以一开始从D[c]开始
70. for(int j = R[i];j != i;j = R[j])
71. v[Col[j]] = false;
72. }
73. return ret;
74. }
75. void Dance(int d)
76. {
77. //if(d + f() >= ansd)return;剪纸，这样会TLE。去掉注释就不TLE了
78. if(R[0] == 0)
79. {
80. if(d < ansd)ansd = d;
81. return;
82. }
83. int c = R[0];
84. for(int i = R[0];i != 0;i = R[i])//找到第c列，还没有被消除的
85. if(S[i] < S[c])
86. c = i;
87. for(int i = D[c];i != c;i = D[i]) //穷举消除
88. {
89. remove(i);//第c列第i行
90. for(int j = R[i];j != i;j = R[j])remove(j); //删除这一行,因为是可重复的，所以值要删这一行的所有元素，同时这一行元素的列也删去
91. Dance(d+1);
92. for(int j = L[i];j != i;j = L[j])resume(j);//恢复
93. resume(i);
94. }
95. }
96. }dlx;
97. char s[200];
98. int mp[20][20], a[20][20];
99. int x, y;
100. int guaishou;
101. void makeit(int p, int row, int col)
102. {
103. for (int i = row; i <= row + x - 1; ++ i)
104. for (int j = col; j <= col + y - 1; ++ j)
105. {
106. if (a[i][j] == 1) dlx.Link(p, mp[i][j]);
107. }
108. }
109. void init()
110. {
111. guaishou=0;
112. for (int i = 1; i <= n;++i)
113. for (int j = 1; j <= m;++j)
114. {
115. scanf("%d", &a[i][j]);
116. if (a[i][j])
117. { ++guaishou;
118. mp[i][j] = guaishou;
119. }
120. }
121. scanf("%d%d", &x, &y);
122. dlx.init(n \* m, guaishou);
123. int sb=0;
124. for (int i = 1; i + x - 1<= n; ++ i)
125. for (int j = 1; j + y - 1 <= m; ++ j)
126. {
127. ++sb;
128. makeit(sb, i, j);
129. }
130. }
131. int main()
132. {
133. int t=0;
134. for (int i = 1; i <= 15;++i)
135. for (int j = 1; j <= 15; ++j)
136. if (mp[i][j] = ++t);
137. while (~scanf("%d%d", &n, &m))
138. {
139. init();
140. dlx.Dance(0);
141. printf("%d\n", dlx.ansd);
142. }
143. return 0;
144. }

## 线段树的几个常见板子

### 一维线段树

整理板子是一维，有一些常见的板子写法。

1. 单点修改，求区间最大值线段树

int val[maxn<<2], mx[maxn<<2];

#define lson o\*2, L, M

#define rson o\*2+1, M + 1,R

int segment\_tree\_query\_l, segment\_tree\_query\_r, segment\_tree\_query\_ans;

//int segment\_tree\_update\_l, segment\_tree\_update\_r, segment\_tree\_update\_val;

//这种线段树写法，需要把询问的东西，放在全局变量

//qans就是query询问的结果，ql,qr是询问[L,R]区间的答案

//ql,qr,qans对于update的时候，本着不浪费的原则，复用一下，就不用upl,upr,和upval了

#define qans segment\_tree\_query\_ans

#define ql segment\_tree\_query\_l

#define qr segment\_tree\_query\_r

//upl ,upr是update的[L,R]区间，改为upval

#define upl segment\_tree\_update\_l

#define upr segment\_tree\_update\_r

#define upval segment\_tree\_update\_val

//提前给ql,qr,qans赋值！单点修改，所以ql==qr。这个点改为qans

void update(int o, int L, int R)// L,R区间统一改为qans

{

if (ql <= L && R <= qr)

{

val[o] = mx[o] = qans;

return ;

}

int M = L + (R - L) / 2;

if (ql <= M) update(lson);

if (qr > M) update(rson);

mx[o] = max(mx[o\*2], mx[o\*2+1]);

}

//提前给qans赋值为最小值！出ql,qr赋值，表示为查询的区间为[ql,qr]

void query(int o, int L, int R)

{

if (ql <= L && R <= qr)

{

qans= max(qans, mx[o]);

return ;

}

int M = L + (R- L)/2;

if (ql <= M) query(lson);

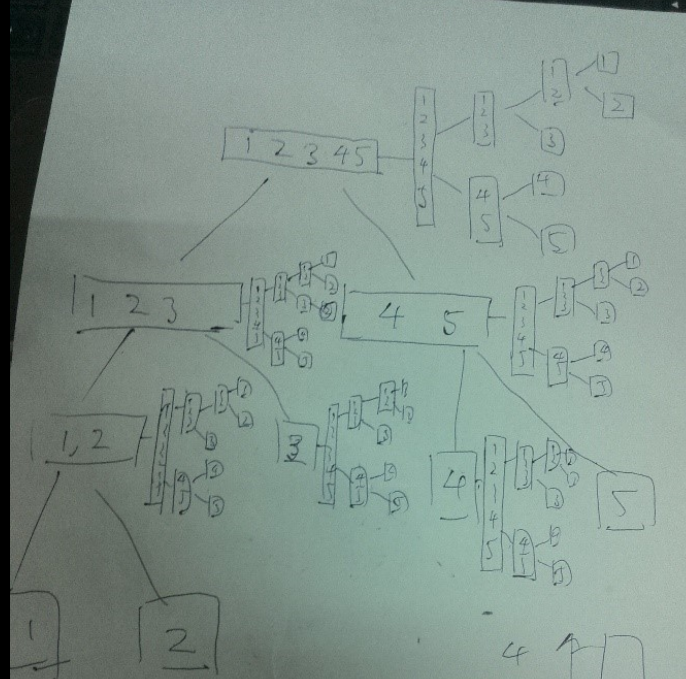
if (qr > M) query(rson);

}

### 二维线段树

1. 二维线段树简介

QC打野说二维线段树不支持打标机。只能标记永久话。这样好啊，这样避免了很多复杂的题。而且，询问通常是单点查询。



上图说明了，二维线段树是干嘛的。没错，就是每次都维护一大堆区间，每个线段树再开一个线段树来维护。这样，每个区间就表示x方向（不是ij坐标系，是xy坐标系）[1,3]区间，并且y方向上[1,5]区间的情况如何如何……

1. 代码实现

*例题1：*二维区间里，某个矩形里都是01， 选一个矩形，里面数字01翻转。最后不停的问某个坐标是0还是1。

#define lson o\*2, L, M

#define rson o\*2+1, M + 1,R

int cov[4001][4001];

int n, m;

int xql, xqr, yql, yqr;

void init()

{

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

memset(cov, 0, sizeof(cov));

}

void y\_update(int k, int o, int L, int R)

{

if (yql <= L && R <= yqr)

{

cov[k][o] ^= 1;

return;

}

int M = L + (R-L)/2;

if (yql <= M) y\_update(k, lson);

if (yqr > M) y\_update(k, rson);

}

void x\_update(int o, int L, int R)

{

//cout<<o<<" "<<L<<" "<<R<<" "<<xql<<" "<<xqr<<endl;

if (xql <= L && R <=xqr)

{

y\_update(o, 1, 1, n);

return;

}

int M = L + (R - L)/2;

if (xql <= M) x\_update(lson);

if (xqr > M) x\_update(rson);

}

int ans;

int qx, qy;

void y\_query(int k, int o, int L, int R)

{

ans ^= cov[k][o];

// if (qy <= L && R <= qy)

if (L ==R)

{

return;

}

int M = L + (R - L)/2;

if (qy <= M) y\_query(k, lson);

else y\_query(k, rson);

}

void x\_query(int o, int L, int R)

{

y\_query(o, 1, 1, n);

//if (qx <= L && R<=qx)

if (L==R)

{

return;

}

int M = L + (R - L)/2;

if (qx <= M) x\_query(lson);

else x\_query(rson);

}

void doit()

{

while (m--)

{

char flag;

//prln(flag);

getchar();

scanf("%c", &flag);

//prln(flag);

if (flag=='C')

{

scanf("%d%d%d%d", &xql, &yql, &xqr, &yqr);

//cout<<xql<<" "<<xqr<<" "<<yql<<" "<<yqr<<endl;

x\_update(1, 1, n);

//cout<<flag<<" "<<xql<<" "<<xqr<<" "<<yql<<" "<<yqr<<endl;

}

else

{

scanf("%d%d", &qx, &qy);

ans = 0;

x\_query(1, 1, n);

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

//cout<<flag<<" "<<qx<<" "<<qy<<endl;

}

}

}

*例子2：*题目大意：给定一个矩形，每次选出一个矩形覆盖上颜色K，最后问你每个颜色有多少个。（后上色的，会覆盖掉之前上色的）。

首先离散化所有坐标，构建二维线段树。因为树套树不能动态标记，只能永久标记，所以我们读入后，把数据倒着处理。先进树的，可以让后进树的进不去。

我们假设树套树一开始做的是X树，后来在X树的节点上，有Y树。对于覆盖区间，找到对应的X树，再找到对应的Y树，如果在找对应Y树的过程中，发现已经被彻底染色的区间，直接return（因为倒着处理了，不能覆盖之前的区间）。

如果没有被彻底染色的区间，则记录上【时间戳】和【颜色】。

查询：最后n^2次，查询每个格点，问每个格点是什么色。

对于问一个格点是什么色，先找到对应的X树，在找X树的过程中，所有经过的X树的节点，都去查询Y树的结果。对于每次结果，如果有时间戳更迟的，则更新答案。

#define lson o\*2, L, M

#define rson o\*2+1, M + 1,R

int cov[4001][4001]={0};//表示区间颜色是什么

int dfn[4001][4001]={0};

int vis[4001]={0};

int xql, xqr, yql, yqr;

int col, row, n;

int colo, id;

struct ls// 离散化

{

short lisan[8000];//最大数字数量

int t; //当前数字数量

void ins(int k) //添加一个数字

{

++t;

lisan[t] =k;//下标从0开始

}

void clear() //初始化所有信息

{

t=0;

}

void doit() //开始离散化

{

sort(lisan + 1, lisan + 1+ t);

t = std::unique(lisan + 1, lisan + 1+ t) - (lisan + 1);

}

int get(int k) //查找k的下标, 确保查询的数字在离散化是数列之内

{

return std::lower\_bound(lisan + 1, lisan + 1 + t, k) - (lisan);

}

int get2(int k) //查找比K大的下标。比如2 3 3 5 ，k=3返回的是5的下标

{

return std::upper\_bound(lisan + 1, lisan + 1 + t, k) - (lisan);

}

int operator [] (int k)

{

return get(k);

}

}lisan;

void y\_update(int k, int o, int L, int R)

{

if (cov[k][o]) return; //如果已经有颜色了，退出

if (yql <= L && R <= yqr)

{

cov[k][o] = colo;

dfn[k][o] = id; //啥时间涂的

return;

}

int M = L + (R-L)/2;

if (yql <= M) y\_update(k, lson);

if (yqr > M) y\_update(k, rson);

}

void x\_update(int o, int L, int R)

{

//cout<<o<<" "<<L<<" "<<R<<" "<<xql<<" "<<xqr<<endl;

if (xql <= L && R <=xqr)

{

y\_update(o, 1, 1, lisan.t - 1);

return;

}

int M = L + (R - L)/2;

if (xql <= M) x\_update(lson);

if (xqr > M) x\_update(rson);

}

int ans, anstime;

int qx, qy;

void y\_query(int k, int o, int L, int R)

{

if (dfn[k][o] > anstime)

{

ans = cov[k][o];

anstime = dfn[k][o];

}

//pr(k),pr(o),pr(L),pr(R),pr(ans),pr(anstime), pr(dfn[k][o]), prln(cov[k][o]);

if (L == R)

{

return;

}

int M = L + (R - L)/2;

if (qy <= M) y\_query(k, lson);

else y\_query(k, rson);

}

void x\_query(int o, int L, int R)

{

//pr(o),pr(L),prln(R);

//cout<<xql<<" -- "<<xqr<<" "<<(xql <= L && R <=xqr)<<endl;

y\_query(o, 1, 1, lisan.t - 1);

if (L == R)

{

return;

}

int M = L + (R - L)/2;

//cout<<(xql <= M)<<" "<<(xqr > M)<<endl;

if (qx <= M) x\_query(lson);

else x\_query(rson);

}

struct node

{

int lx, ly, rx, ry, colo;

}sq[1000+5];

int output[3000+5];

void doit()

{

// cout<<"#"<<endl;

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

{

xql = lisan.get(sq[i].lx);

yql = lisan.get(sq[i].ly);

xqr = lisan.get(sq[i].rx) - 1;

yqr = lisan.get(sq[i].ry) - 1;

// pr(xql),pr(yql),pr(xqr),prln(yqr);

colo = sq[i].colo;

id = i;

x\_update(1, 1, lisan.t - 1);

}

/\*

while (1)

{

qx=4,qy=4;

//cin >> qx>>qy;

ans=-1;

anstime=-1;

x\_query(1,1, lisan.t - 1);

pr(qx),pr(qy),prln(ans);

//cout<<ans<<endl;

exit(0);

}

\*/

// cout<<"@"<<endl;

// exit(0);

memset(output, 0, sizeof(output));

for (int i = 1; i < lisan.t; ++ i)

for (int j = 1; j < lisan.t; ++ j)

{

qx = i;

qy = j;

ans = -1;

anstime = -1;

x\_query(1, 1, lisan.t - 1);

//cout<<endl<<endl<<endl;

// pr(qx),pr(qy),prln(ans);//,pr(lisan.lisan[i]),pr(lisan.lisan[i+1]),pr(lisan.lisan[j]),prln(lisan.lisan[j+1]);

// continue;

if (ans ==-1)

{

cout<<"su bi it"<<endl;

exit(0);

}

int chang = lisan.lisan[i + 1] - lisan.lisan[i];

int kuan = lisan.lisan[j + 1] - lisan.lisan[j];

output[ans] += chang \* kuan;

}

for (int i = 0; i <= 2500; ++ i)

if (output[i])

printf("%d %d\n", i, output[i]);

}

int main()

{

scanf("%d%d%d", &row, &col, &n);

lisan.clear();

lisan.ins(row);

lisan.ins(col);

lisan.ins(0);

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

{

int lx, ly, rx, ry, colo;

scanf("%d%d%d%d%d", &lx, &ly, &rx, &ry, &colo);

// cout<<lx<<" "<<ly<<" "<<rx<<" "<<ry<<" "<<colo<<endl;

//lx++;

//ly++;

sq[i] = {lx, ly, rx, ry, colo};

lisan.ins(lx);

lisan.ins(ly);

lisan.ins(rx);

lisan.ins(ry);

}

sq[0] = {0, 0, row, col, 1};

//for (int i = 1; i <= lisan.t;++i)

// cout<<lisan.lisan[i]<<" ";cout<<endl;

lisan.doit();

//for (int i = 1; i <= lisan.t;++i)

// cout<<lisan.lisan[i]<<" ";cout<<endl;

doit();

return 0;

}

/\*

20 20 3

2 2 18 18 2

0 8 19 19 3

8 0 10 19 4

1 91

2 84

3 187

4 38

\*/

### 有序表线段树

1、有序表介绍

所谓有序表线段树，就是线段树的每个区间保存的序列是有序表（这可能有点废内存）。

比如数组为：5 4 1 7 2，[1,5]区间内， 还保存了一个1 2 4 5 7的有序数组。在[1,3]内，保存一个[1, 4,5]的有序数组。有序数组可以用归并排序快速得到（在构建线段树的时候就顺便得到了）。

下程序主要实现的是，对b数组进行构造有序表线段树。

1. //对b数组构造有序表
2. void build(int o, int L, int R,int deep)
3. {
4. if (L == R)
5. {
6. ans[o] = a[L] >= b[L];
7. st[deep][L] = b[L];
8. return ;
9. }
10. int M = L + (R -L)/2;
11. int lc = o \* 2, rc = o \* 2 + 1;
12. build(lc, L, M, deep + 1);
13. build(rc, M + 1, R, deep + 1);
14. ans[o] = ans[lc] + ans[rc];//区间有多少个b[i]>a[i]
15. lc=L, rc= M + 1;//lc,rc分别对应左右的开始节点
16. for (int i = L; i <= R; ++ i) //归并排序
17. if (rc > R || lc <= M & st[deep + 1][lc] <= st[deep + 1][rc])
18. {
19. st[deep][i] = st[deep + 1][lc++];
20. }
21. else
22. {
23. st[deep][i] = st[deep + 1][rc++];
24. }
25. //st[deep][i]表示第i层的i位置的数字
26. //下面的代码并不是有序表相关的了。
27. lc = L, rc = M + 1;
28. for (int i = L; i <= R; ++ i) //计算每个数字在左右儿子中的情况
29. {
30. while (st[deep][i] > st[deep + 1][lc] && lc <= M) ++lc;
31. lk[deep][i] = lc;
32. while (st[deep][i] > st[deep + 1][rc] && rc <= R) ++ rc;
33. rk[deep][i] = rc;
34. }
35. }

当然线段树的写法可以套用之前的线段树写法，来实现更小的常数。有空进行修改。

有序表线段树，可以实现一些需要的功能。下面用一个例子说明。

2、例题详解

例题：HDU 5737 2016多校的题

题目大意：

给a[ ],b[ ]两个数组。有2个操作。

操作1：把a数组[L,R]区间全部修改为x

操作2：询问[L,R]区间，ai>=bi的数量。

比如[4,7]， 答案就是 (a4>=b4 ) + (a5>=b5) + (a6>=b6) + (a7>=b7)【为bool运算，结果返回0,1……】

算法1：

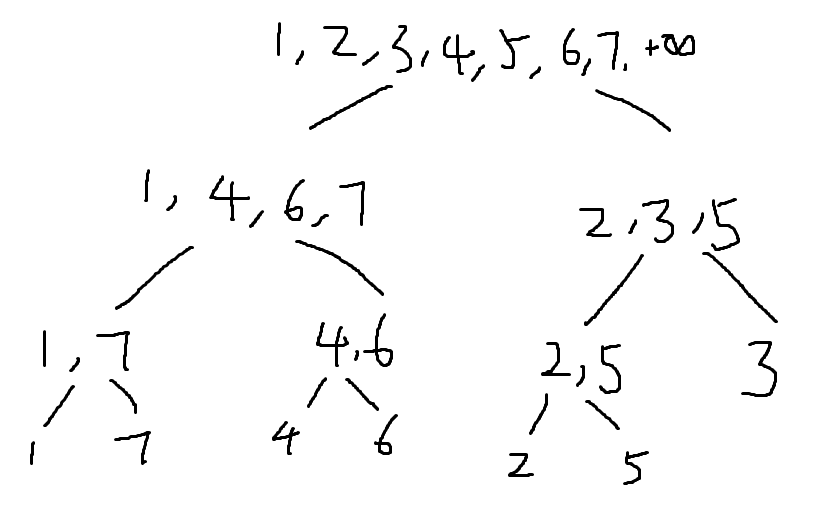
以b数组构建有序表线段树。除此之外，还维护每个节点的ans函数，表示[L,R]区间满足ai>bi的数量。

对于修改操作（题目只会修改a数组，并且一段都改为x），我们只需要找到线段树[L,R]区间，然后在有序表区间里二分找一下x。就可以更新ans函数了。但是因为多了二分查找，所以速度较慢，这题会TLE。

算法2：

按照方法1的思想，我们主要维护的就是线段树上的ans这个值了。只要线段树上ans的值是对的，就可以方便的query了。

举一个简单的线性表线段树的例子



如上图所示，假如我们需要修改[3,5]区间都为5 。那么在[1,7]区间比5大的最小的数字是6。现在就是在每个子区间找大于等于6的数字了。

在左区间大于等于 6的数字依然是6（下标为4），在右区间则没有这样的数字。那么在右区间直接记录下标为8（右区间是[5,7],8的话表示超出区间标识）。

用类似这样的方法递归下去，就可以动态的修改线段树维护的ans域的信息了。AC code见附录。

四、

### 附录

1、HDU 5737 AC CODE

1. typedef long long LL;
2. const int maxn = 10e5 \*4 + 10;
3. const LL mod = 1e9 + 7;
4. int a[maxn], b[maxn];
5. int n, m;
6. int ans[maxn], paixu[maxn], tmp\_array[maxn];
7. int st[20][maxn], lk[20][maxn], rk[20][maxn];//排序后的数组,每一层的若干个位置
8. int change[maxn];
9. #define lson o\*2, L, M
10. #define rson o\*2+1, M + 1,R
11. #define self o, L, M
12. namespace getnum
13. {
14. int a , b , C = ~(1<<31), M = (1<<16)-1;
15. inline int rnd(int last) {
16. a = (36969 + (last >> 3)) \* (a & M) + (a >> 16);
17. b = (18000 + (last >> 3)) \* (b & M) + (b >> 16);
18. return (C & ((a << 16) + b)) % 1000000000;
19. }
20. };
21. using getnum::rnd;
22. //对b数组合并
23. void build(int o, int L, int R,int deep)
24. {
25. if (L == R)
26. {
27. ans[o] = a[L] >= b[L];
28. st[deep][L] = b[L];
29. return ;
30. }
31. int M = L + (R -L)/2;
32. int lc = o \* 2, rc = o \* 2 + 1;
33. build(lc, L, M, deep + 1);
34. build(rc, M + 1, R, deep + 1);
35. ans[o] = ans[lc] + ans[rc];//区间有多少个b[i]>a[i]
36. lc=L, rc= M + 1;//lc,rc分别对应左右的开始节点
37. for (int i = L; i <= R; ++ i) //归并排序
38. if (rc > R || lc <= M & st[deep + 1][lc] <= st[deep + 1][rc])
39. {
40. st[deep][i] = st[deep + 1][lc++];
41. }
42. else
43. {
44. st[deep][i] = st[deep + 1][rc++];
45. }
46. //st[deep][i]表示第deep层的i位置的数字
47. //下面的代码并不是有序表相关的了。
48. //lk[deep][i]表示第deep层，排第i名的数字，在左儿子层理，可以排第几
49. //rk同理，在右儿子里的排名
50. lc = L, rc = M + 1;
51. //这里lc,rc变量复用，不浪费变量而已。
52. for (int i = L; i <= R; ++ i) //计算每个数字在左右儿子中的情况
53. {
54. while (st[deep][i] > st[deep + 1][lc] && lc <= M) ++lc;
55. lk[deep][i] = lc;
56. while (st[deep][i] > st[deep + 1][rc] && rc <= R) ++ rc;
57. rk[deep][i] = rc;
58. }
59. }
60. int ql, qr, v;
61. inline void push\_down(int o, int L, int R, int deep)
62. {
63. //change[o]不为-1，说明有消息需要push\_down
64. if (change[o] != -1 && L < R)
65. {
66. int lc = o \* 2, rc = o \* 2 + 1, M = L + (R-L)/2;
67. change[lc] = change[o] > R ? M + 1 : lk[deep][change[o]];
68. change[rc] = change[o] > R ? R + 1 : rk[deep][change[o]];
69. change[o] = -1;
70. }
71. }
72. inline void maintain(int o, int L, int R)
73. {
74. int lc = o \* 2, rc = o \* 2 + 1;
75. if (change[o] != -1) ans[o] = change[o] - L;
76. else if (L < R) ans[o] = ans[lc] + ans[rc];
77. }
78. int query(int o, int L, int R,int deep) //询问ql,qr区间
79. {
80. if (change[o] != -1) //有标记的情况下，输出标记信息
81. {
82. //有些题可以直接根据段标记知道答案
83. //就可以直接return了
84. //return change[o] - L;
85. }
86. if (ql <=L && R <= qr)
87. {
88. if (change[o] != -1) ans[o] = change[o] - L;
89. return ans[o];
90. }
91. push\_down(self, deep);
92. int M = L + (R - L)/2;
93. int ret = 0;
94. if (ql <= M) ret += query(lson, deep + 1);
95. else maintain(lson);
96. if (qr > M) ret += query(rson, deep + 1);
97. else maintain(rson);
98. maintain(self);
99. return ret;
100. }
101. void update(int o, int L, int R, int deep, int flag)
102. //给ql,qr区间，都变为[不超过第flag的最大数字]，其中ql,qr是全局变量
103. //小trick,对于一个数字大的超过区间所有数字，那么返回的就是[L,R]的R+1
104. {
105. if (ql <= L && R <= qr)
106. {
107. change[o] = flag;
108. ans[o] = flag - L;
109. return;
110. }
111. else
112. {
113. push\_down(self, deep);
114. int M = L + (R - L)/2;
115. if (ql <= M) update(lson, deep + 1, flag >R ? M + 1 : lk[deep][flag]);
116. else maintain(lson);
117. if (qr > M) update(rson, deep + 1, flag > R ? R + 1 : rk[deep][flag]);
118. else maintain(rson);
119. }
120. maintain(self);
121. }
122. inline void UD(int L, int R, int x)
123. {
124. //L,R区间都改为X
125. int l = 0, r = n + 1, m;
126. st[1][n + 1]=0x7fffffff;//额外添加一个INF的数字，保证总有最大的数字
127. while (l + 1 < r) //(l,r]区间找比x大的最小值
128. {
129. m = l + (r-l)/2;
130. if (st[1][m]<=x) l = m;
131. else r = m;
132. }
133. //r为[L,R]中，比x大的最小值
134. update(1, 1, n, 1, r);
135. }
136. void init()
137. {
138. scanf("%d%d%d%d", &n, &m, &getnum::a, &getnum::b);
139. for (int i = 1; i <= n; ++ i) scanf("%d", &a[i]);
140. for (int i = 1; i <= n; ++ i) scanf("%d", &b[i]);
141. build(1,1,n,1);
142. }
143. void doit()
144. {
145. int last =0;
146. LL ans = 0;
147. memset(change , -1, sizeof(change));
148. for (int i = 1; i <= m; ++ i)
149. {
150. int l = rnd(last) % n + 1;
151. int r = rnd(last) % n + 1;
152. int x = rnd(last) + 1;
153. if (l>r) swap(l,r);
154. ql = l, qr = r;
155. if ((l+r+x)%2==0)
156. {
157. //询问
158. last = query(1, 1, n, 1);
159. ans = (ans + 1LL \* i \* (LL)last) % mod;
160. }else
161. {
162. //修改
163. UD(l, r, x);
164. }
165. }
166. printf("%lld\n", ans);
167. }
168. int main()
169. {
170. int T;
171. scanf("%d", &T);
172. while (T--)
173. {
174. init();
175. doit();
176. }
177. return 0;
178. }

2、HDU 5957的线段树部分,使用数组模拟指针的方式来完成线段树，动态维护区间和。（区间整体增加一个数字，区间整体减少一个数字）

1. const int maxnode = ???;

2.

3. struct node

4. {

5. int lson, rson;

6. int val;//区间权重

7. int change;//区间修改值,为0表示没有需要传递的

8. }tree[maxnode];

9. int tail;

10.

11. int ql, qr, qans;

12. #define LSON tree[o].lson,L, M

13. #define RSON tree[o].rson, M + 1 , R

14. #define SELF o,L,R

15. #define lc tree[o].lson

16. #define rc tree[o].rson

17.

18. void build(int o, int L, int R)

19. {

20. tree[o].change = 0;

21. tree[o].val = 0;

22. if (L== R)

23. {

24. tree[o].val = 0;

25. return ;

26. }

27. tree[o].lson = ++ tail;

28. tree[o].rson = ++ tail;

29. int M = L + (R - L) / 2;

30. build(LSON);

31. build(RSON);

32. }

33.

34. void push\_down(int o, int L, int R)//向下传递标记

35. {

36. if (tree[o].change && L < R)

37. {

38. tree[o].val += tree[o].change \* (R-L+1);

39. tree[lc].change += tree[o].change;

40. tree[rc].change += tree[o].change;

41. tree[o].change = 0;

42. }

43. }

44.

45. void maintain(int o, int L, int R)//把o的儿子的信息，归到o节点

46. {

47. push\_down(SELF);

48. if (L < R)

49. {

50. int M = L+(R-L)/2;

51. //去获取儿子的信息更新到父亲

52. //儿子节点的信息，要通过和儿子节点的change计算得到

53. tree[o].val = tree[lc].val + tree[rc].val + tree[lc].change \* (M-L+1) + tree[rc].change \* (R-M);

54. }

55. }

56.

57. int query(int o, int L, int R)

58. {

59. if (ql <= L && R <= qr)

60. {

61. //要考虑change的信息

62. return tree[o].val + tree[o].change \* (R-L+1);//因为根节点的传递标记一直都在

63. }

64. push\_down(SELF);

65. int M = L + (R - L) / 2;

66. int ret = 0;

67. if (ql <= M) ret += query(LSON);

68. else maintain(LSON);

69. if (qr > M) ret += query(RSON);

70. else maintain(RSON);

71. maintain(SELF);

72. return ret;

73. }

74.

75. void update(int o, int L, int R)

76. {

77. if (ql <= L && R <= qr)

78. {

79. tree[o].change += qans;

80. return;

81. }else

82. {

83. push\_down(SELF);

84. int M = L + (R-L)/2;

85. if (ql <= M) update(LSON);

86. else maintain(LSON);

87. if (qr > M) update(RSON);

88. else maintain(RSON);

89. }

90. maintain(SELF);

91. }

92.

93. void doit()

94. {

95. tail=0;

96. build(0,1,n);

97. }

## 斜率DP

### 常见套路

对于形如这样的方程



其中是常数，为定值。数组为单调递增且非负的数组，也为常数。

这样的DP方程的求解需要的时间解决。很多时候这是不可接受的。所以我们要寻求一个合理的优化方案。

对于上述方程，如果, 从和转移的结果分别如下



如果从转移会比从转移更优，那么必然满足如下条件（上面的式子减去下面的式子大于0的情况）大致得到形如这样的式子。为单调递增并且大于0，所以显然都是大于0并且单调递增的。

形如上述的式子，可以再写为

可以看出来，如果   (i>j>k) 的话，那么j就是废点，因为如果,那么i比j优，不管是多少，i都比j优。那么j就是废点了。

当的时候， 那么 ，说明,那么说明k比j要优。那么j就废了。

所以任何相邻的编号，他们连线的斜率，一定是单调递增的（所以去掉了很多无用点）。而解，一定是选择其中一个点。这样的点组成，可以显然的看出，是一个下凸壳。

也就是看起来像y=x^2 在第一象限的图案( 显然事实的图案是离散的。。。并且是一顿一段的，不可能光滑连续哒！)

【可能出现2个点因为巧合，所以重叠！】

维护凸壳，显然就用一个栈可以实现啦。

求解呢？ 如果sum是无序的，那么就比较麻烦（平衡树之类的）

考虑sum=p的情况， 对于所有的凸壳上的点，从左下角第一个点开始往右找，斜率在提升，所以只要斜率小于p，那么就是我们需要的转移的点。

比如斜率依次为 1 3 4 5 6 7 9 10 11 15， P=8时，则会选择斜率为g(q[7],q[6]) = 7 (第7个点，和第6个点的斜率为7)，然后q[7]显然更优，因为满足g(q[7],q[6])<sum.

同理，q[7]比q[8]优，因为g(q[8],q[7])>sum

又因为sum是单调递增的（不单调递增真的前面推导很多都要分类讨论啦！我也不想考虑了……就按照单调递增来算），

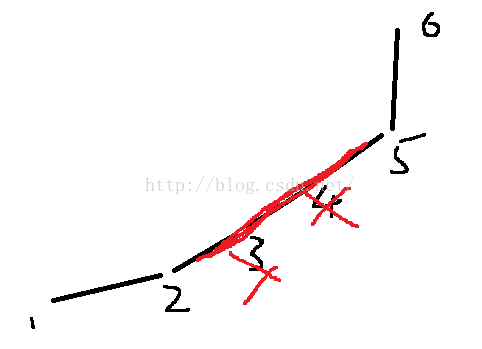
所以如果选过q[7],那么q[1...6]都不可能再次选择了。 这和sum是单调增有关系，证明略，稍微想一想就能想出来了……

然后要考虑一个情况，就是3个点斜率相同~ 这都是程序里考虑的问题了。

【比较2个斜率大小，不要比斜率！要化简公式避免除法，除法会溢出（除0）！！！】

【PS：关于如果存在比较符号的>= >是否取等于号的问题】

如果存在斜率如图所示：



我们3,4会出队。

如果程序中去掉等号的话……（源程序是有等于号的）

bool slope(int i, int j, int k) //得出j,k斜率

{

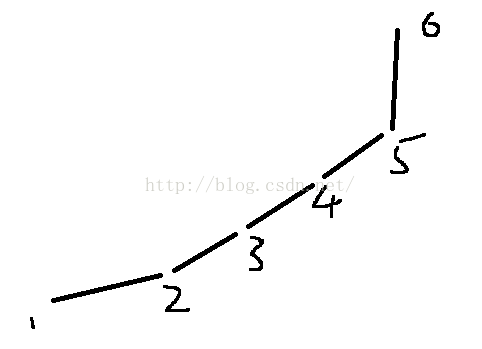
//判断i,j的斜率，和j,k的斜率谁更大

//如果i,j斜率<=j,k斜率，那么就需要退了

return (y(i) - y(j)) \* (x(j) - x(k)) <= (y(j) - y(k)) \* (x(i) - x(j)); //这一行的等于号去掉的话

}

如果没有等于号，就会得到如下图所示的情况



看似没有问题？是不是只要在求解的过程中注意判断就行了呢？

因为，一开始的假设中，我们假设yi = f[i] - sum[i]^2, x[i] = sum[i] 或者x[i] = 2 \* sum[i]

如果题目给的数字a[i]有0的话，那么sum[i]不变，但是f[i]显然是非递减的。也就是说，可能出现若干个点的x坐标相同，

（上述都是其他重点。。如果不是我用转移结果来判断优劣的话……）

实际上的真正导致错误的原因，是可能先从2转移，再从3转移，再从4转移，再从5转移。导致M被加了很多次……

比如这组数据~~~

4 1

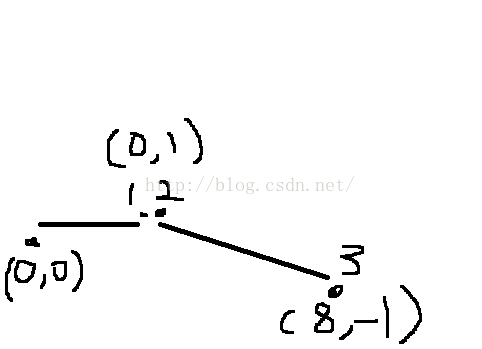
0

0

4

1

在3个点的时候，出现这样的情况



1,2两个点，因为sum为0,的原因，重叠了。这个时候，3应该操死1,2，然后直接连0. 但是因为1,2的斜率不存在，所以出现了遗漏，导致3并没有操死1,2的任何一个点。

这样，就导致出现了问题。也就是说，如果出现【重复点】的情况下，会导致错误。为了避免这种情况，我们在程序的slope函数中，就加上了小于等于号。这样就去掉了重复点，和斜率相同的点之类的情况。 当然，既然没有重复点，实际上

while (head + 1 != tail && zhuan(q[head], i) >= zhuan(q[head + 1], i)) //如果队列里元素超过1个，那么就比较

++head;

这一段中的>=号，改为 >也是可以的了。

貌似没有疑问了……

AC CODE

#include <cstdio>

#include <iostream>

#include <cstdlib>

#include <cstring>

using namespace std;

typedef long long LL;

const int maxn = 500000 + 100;

int n, m;

LL a[maxn], s[maxn];

LL q[maxn], tail, head;

LL f[maxn];

const LL inf = 1LL<<50;

LL y(int k)

{

return f[k] + 1LL \* s[k] \* s[k];

}

LL x(int k)

{

return 2LL \* s[k];

}

bool slope(int i, int j, int k) //得出j,k斜率

{

//判断i,j的斜率，和j,k的斜率谁更大

//如果i,j斜率<=j,k斜率，那么就需要退了

return (y(i) - y(j)) \* (x(j) - x(k)) <= (y(j) - y(k)) \* (x(i) - x(j));

}

LL zhuan(int j, int i)//j<i,j向i转移的值

{

return f[j] + (s[i] - s[j]) \* (s[i] - s[j]) + m;

}

void init()

{

tail = head = 0; //tail == head则为空

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

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

for (int i = 1; i <= n; ++ i) s[i] = s[i - 1] + a[i];

q[tail++] = 0;

}

void doit()

{

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

{

while (head + 1 != tail && zhuan(q[head], i) >= zhuan(q[head + 1], i)) //如果队列里元素超过1个，那么就比较

++head;

f[i] = zhuan(q[head], i);

while (head + 1 != tail && slope(i, q[tail - 1], q[tail - 2]))

--tail;

q[tail++] = i;

}

printf("%lld\n", f[n]);

}

int main()

{

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

{

init();

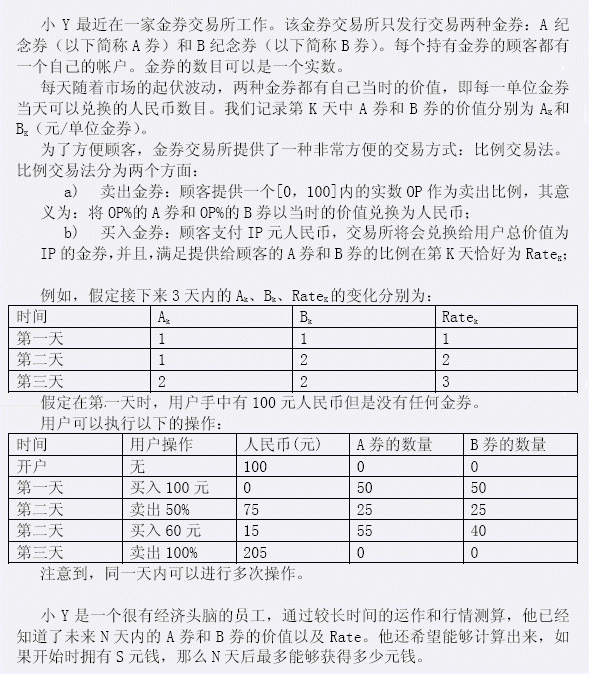
doit();

}

return 0;

}

NOI货币兑换



f[i]表示第i天，所能获得的最大 金钱数量。

如果知道了f[i],那么我们可以用x[i]表示第i天的钱，全部买A卷，能购买的数量，y[i]则为B卷数量。

至于x[i], y[i]的求解，就不再赘述……（列方程表示一下就能求出来啦~）

然后f[i]可以写成类似于 f[i] = max{f[i-1], max{x[j]\*a[i]+y[j]\*b[i]}}的形式

不考虑狮子中f[i-1] 的部分，强行变换一下

f[i]=x[j]\*a[i]+y[j]\*b[i]

式子变形后为

f[i]/b[i] - x[j]\*(a[i]/b[i])=y[j]

a[i],b[i]为常数，也就是第i天A,B卷的价格。

这个式子形如y=kx+b

求i的时候，[1,i-1]的f值都已经知道了，所以可以弄出很多x[j],y[j]的点对。 相当于我在一张图上，有很多（x，y）的坐标，我有一个斜率k，要找一个点，去放上这个斜率k，使得方程中的b最大。

y=kx+b是截距式，并且k恒为负数（a[],b[]为正数，式子中有符号。 同时，题目给定的rate比例是[0,100]，保证了b为非0，所以斜率永远存在~）

现在问题就转变为，根据前面的乱七八糟的x,y的点坐标，找一个最优的点坐标，来匹配上k，求出f[i]的值啦

然后显然（这个显然可以自己画图……当然别人的文章有图啦~我比较懒） 解一定在上凸壳上，所以首先我们要维护一个上凸壳。

对于每次插入新的点，可以看这个点在凸壳内部（下方），还是上方。因为凸壳的点 x坐标是有序的，所以可以用map维护x坐标，和x坐标里保存的y坐标信息，以及这个点和左边的点的斜率，和右边点的斜率。

比如说，第三个点，和第二个点斜率， 以及和第四个点的斜率。维护这些信息会方便一些~

【第一个点和左边的点斜率为0， 最后一个点和右边的点斜率为负无穷】 这个很重要，问我为啥知道的。。我举例子弄的……没法证明。

然后再用一个map来维护斜率，映射斜率对应的x坐标即可。

然后维护这两个map,我的代码太臃肿，并且因为偷懒，反复调用了很多东西。而且确实，我的map水平不堪入目啊……

然后细节很多。。。

情况1： 插入的点，其x坐标是已插入点最大/最小的。

情况2：插入的点，和之前插入的点，x坐标相同（若新点y坐标更小，直接抛弃，否则需要删除曾经插入的点，来插入新的点）

还有一些情况，在写map的时候回自己感觉到。。。如果写 splay，就不会因为出现不好判断(--map.begin())之类的情况的问题啦~ 其实写其他平衡树也行。

CDQ分治代码如下：

#include<iostream>

#include<cstdio>

#include<cstdlib>

#include<algorithm>

#include<cstring>

#define ll long long

using namespace std;

const int maxn = 100005;

const double inf = 1e12;//-inf就是负无穷了

const double eps = 1e-9;

double f[maxn]={0};

struct node

{

int id;

double x, y, k;

double a, b, r;

}p[maxn], t[maxn];

node stk[maxn];

int top, n;

bool operator <(node A, node B)//斜率倒着排序，为了在凸壳上计算方便

{

return A.k > B.k;

}

bool check(node A, node B, node C)// A,B的斜率如果<=B,C斜率，则为true

{

double a=(B.y-A.y)\*(C.x-B.x);

double b=(C.y-B.y)\*(B.x-A.x);

if (a==b) return true;

if (a>0 && b>0) return a>b;

if (a<0 && a<0) return a<b;

if (a<0) return true;

return false;

}

double xie(node A, node B)

{

if (A.y==B.y) return inf;

return (A.y-B.y)/(A.x-B.x);

}

double getDP(int k, node A)

{

return p[k].a\*A.x +p[k].b\*A.y;

}

void solve(int l, int r)

{

if (l == r)

{

f[l]=max(f[l],f[l-1]);

p[l].y=f[l]/(p[l].a\*p[l].r+p[l].b);

p[l].x=p[l].r\*p[l].y;

return;

}

int mid = l + (r-l)/2;

int l1 =l, l2 = mid + 1;

for (int i = l; i <=r; ++ i)

if (p[i].id <= mid) t[l1++] = p[i];

else t[l2++] = p[i];

for (int i = l; i <= r; ++ i) p[i] = t[i];

solve(l, mid);

top = 0;//[a[top]为栈顶元素，top=0表示空栈，栈内有0个元素。]

//开始做一次凸壳，之前的所有的点，都是按照x坐标排序好了,因为已经不需要求解其f值，已经不需要管顺序了

node tmp;

for (int i = l; i<= mid; ++ i)

{

while (top > 1 && check(stk[top- 1], stk[top], p[i])) top--;

stk[++top] = p[i];

}

tmp;

tmp.x=0;

tmp.y=0;

stk[++top]=tmp;//为了让最后一个节点出现异常，即位最后一个斜率巨大无比是正数。而递减的序列出现正数，不可能更优

int pos=1;

for (int i = mid + 1; i <= r; ++ i)

{

while (pos <=top && getDP(i, stk[pos]) <= getDP(i, stk[pos+1])) pos++;

f[p[i].id] = max(f[p[i].id], getDP(i, stk[pos]));

}

solve(mid + 1, r);

l1=l, l2=mid+1;

//现在已经计算完l,r区间的所有值，并且实际上，已经都取得最优解了。需要给他们按照x坐标，归并排序了

for (int i = l; i <= r; ++ i)

if (l2 > r|| l1 <=mid && (p[l1].x < p[l2].x || p[l1].x==p[l2].x && p[l1].y>p[l2].y)) t[i] = p[l1++];

else t[i] = p[l2++];

for (int i = l; i <=r; ++ i) p[i] = t[i];

}

int main()

{

freopen("cash.in","r",stdin);

freopen("cash.ans","w",stdout);

scanf("%d%lf",&n, &f[0]);

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

{

scanf("%lf%lf%lf", &p[i].a, &p[i].b, &p[i].r);

p[i].id = i;

p[i].k = -p[i].a/p[i].b;

if (p[i].a==0) p[i].k=0;

}

sort(p + 1, p + 1 + n);

solve(1, n);

printf("%.3lf\n", f[n]);

return 0;

}

判断2个点斜率差，可以用我的方法更好一些~

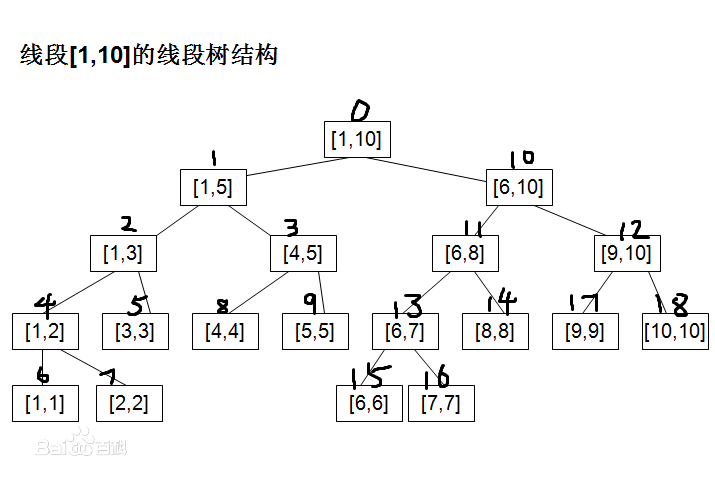
## 主席树

### 线段树

1、线段树介绍

主席树是建立在线段树的基础上的，所以我们首先要介绍线段树。

线段树往往是处理线段区间信息的。



上图表示的就是一颗线段树。我们给每个节点标号。随着构树的顺序，每个节点会有编号。因为个人习惯，我习惯用上述方法给线段树进行编号（给左右儿子编号后，去递归进左儿子，进行左右儿子编号……）

2、线段树代码

下面是常见的线段树带左右儿子指针的写法，当然还有不带左右儿子指针写法的线段树，但是对于主席树，还是需要以这种写法的线段树为蓝图的。

线段树主要分为建树，更新，查询操作。确保线段树每一个节点，都可以立即通过懒人标记与其他参数得到那个节点的权重。其中为了维护懒人标记，需要引入push\_down和maintain操作。

Push\_down(o):向左右儿子传递懒人标记，除此之外不做其他修改，也就是不更新o节点的权重，只是标记传递下去了。

Maintain(o)：维护o节点的权重，从左右儿子获取来更新o节点的权重。特别注意，左右儿子的权重需要考虑左右儿子节点上的懒人标记。

在进入一个节点的时候，如果有进入左右节点的可能的时候，就先push\_down，确保标记能下传下去。如果不进入左儿子，那么就需要maintain(LSON)，同样的不进右儿子那么就要maintain(RSON)。在回溯退出一个节点的时候，也要maintain(o)一下，来确保o节点的权重在退出的时候是正确的。

struct node

{

int lson, rson;

int val;//区间权重

int change;//区间修改值,为0表示没有需要传递的

}tree[maxnode];

int tail;

int ql, qr, qans;

#define LSON tree[o].lson,L, M

#define RSON tree[o].rson, M + 1 , R

#define SELF o,L,R

#define lc tree[o].lson

#define rc tree[o].rson

//线段树需要先建构左右指针一棵树，顺便初始化所有节点的标记信息（change）和权重信息(val)

//懒人标记+val的消息，足够计算出线段树一个节点的值

void build(int o, int L, int R)

{

tree[o].change = 0;

tree[o].val = 0;

if (L== R)

{

tree[o].val = 0;

return ;

}

tree[o].lson = ++ tail;

tree[o].rson = ++ tail;

int M = L + (R - L) / 2;

build(LSON);

build(RSON);

}

//把o节点的懒人标记向下传递，但是并不需要修改o节点的val

void push\_down(int o, int L, int R)//向下传递标记

{

if (tree[o].change && L < R)

{

tree[lc].change += tree[o].change;

tree[rc].change += tree[o].change;

tree[o].change = 0;

}

}

//更新o节点的val信息。那么就需要把儿子的信息汇总到o

//要特别注意，儿子的懒人标记的信息和val信息汇总

void maintain(int o, int L, int R)

{

if (L < R)

{

int M = L+(R-L)/2;

tree[o].val = tree[lc].val + tree[rc].val + tree[lc].change \* (M-L+1) + tree[rc].change \* (R-M);

}

//L==R的情况，不包含的左右儿子，所以再query里询问到的节点，直接结合处理了

}

int query(int o, int L, int R)

{

if (ql <= L && R <= qr)

{

//返回的值需要考虑到懒人标记信息

return tree[o].val + tree[o].change \* (R-L+1);//因为根节点的传递标记一直都在

}

push\_down(SELF);

int M = L + (R - L) / 2;

int ret = 0;

if (ql <= M) ret += query(LSON);

else maintain(LSON);

if (qr > M) ret += query(RSON);

else maintain(RSON);

maintain(SELF);

return ret;

}

void update(int o, int L, int R)

{

if (ql <= L && R <= qr)

{

tree[o].change += qans;

return;

}else

{

push\_down(SELF);

int M = L + (R-L)/2;

if (ql <= M) update(LSON);

else maintain(LSON);

if (qr > M) update(RSON);

else maintain(RSON);

}

maintain(SELF);

}

### 二、主席树

1、主席树介绍

所谓主席树，就是可持久化线段树。所谓可持久化，就是有历史版本的线段树。假设root[i]为第i次操作的线段树的根，那么root[i+1]也就是i+1此操作线段树的根。很多时候，两颗线段树有大量的节点是一样的，这时候就不需要花费额外的空间和时间去处理不一样的地方了，只需要把很多东西复制一下就行了。

举个例子，现在有root[5]以前的所有线段树，我们第六次的时候使用的线段树就是root[6]，一开始root[6]与root[5]是一样的。复制一下线段树的root[5]节点的所有信息，让root[6]的节点与root[5]节点的信息一模样，然后每次操作每到一个节点，就复制这个节点的信息，然后再做修改，这样就完成了带有历史版本的线段树。

2、代码

核心的部分就是insert操作了，build是不变的，因为要建立root[0]的东西。

void build(int o, int L, int R)

{

tree[o].change = 0;

tree[o].val = 0;

if (L== R)

{

tree[o].val = 0;

return ;

}

tree[o].lson = ++ tail;

tree[o].rson = ++ tail;

int M = L + (R - L) / 2;

build(LSON);

build(RSON);

}

void insert(int &o, int L, int R)

{

tree[++ tail] = tree[o];

o = tail;

if (ql <= L && R <= qr)

{

tree[o].change += qans;

return;

}

push\_down(SELF);

int M = L + (R-L)/2;

if (ql <= M) insert(LSON);

else maintain(LSON);

if (qr > M) insert(RSON);

else maintain(RSON);

maintain(SELF);

}

int root[maxn];//表示第i个线段树的root是什么，显然root[0]=0

tail = 0;

build(0,1,n);

root[0]=0;

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

{

root[i] = root[i - 1];

ql = qr = ??;

qans = ??;

insert(root[i], 1, n);

}

3、有动态修改的主席树

用SPOJ TTM为例，说明这些情况。

题目大意：给定一些数字，有以下几个操作

a、[L,R]区间的数字变化D的数值。并且时间向前推进

b、询问某个时间的[L,R]区间的数字之和，这不花费时间

c、回到X时间。

（1）建树

动态修改的时候，会额外开辟很多空间。显然，一开始我们依旧可以给root[0]=0，先构造一颗0版本的版本树，对应0时间的线段树。

struct node //线段树的定义

{

int lson, rson;

long long val;//区间权重

long long change;//区间修改值,为0表示没有需要传递的

int ver;

}tree[maxnode];

int tail, root[maxn];

long long w[maxn];

void init()

{

for (int i = 1; i <=n; ++ i) scanf("%lld", &w[i]);//读入一些数据

root[0] = 0;

tail = 0;

build(root[0], 1, n);

}

void build(int o, int L, int R)

{

tree[o].change = 0;

tree[o].ver = 0;//一开始建树，建的是0号版本的

if (L== R)

{

tree[o].val = w[L];

return ;

}

tree[o].lson = ++ tail;

tree[o].rson = ++ tail;

int M = L + (R - L) / 2;

build(LSON);

build(RSON);

maintain(SELF);

}

上述代码可以看出，和之前的变化是多了ver这个参数，也就是版本号。线段树每个节点，要记录自己是属于哪个版本号的。

(2)修改操作

修改操作，首先是时间的推进，所以根节点要变化。不仅要时间戳增加1（记录当前时间），还有一些细节上不太一样。

++now;//增加时间戳

root[now] = root[now - 1];

qver = now;//全局变量，当前操作的版本号

scanf("%d%d%d", &ql, &qr, &qans);//ql,qr区间的数值变化qans

insert(root[now], 1, n);

其中插入的具体函数如下

void insert(int &o, int L, int R)

{

tree[++ tail] = tree[o];

o = tail;

tree[o].ver = qver;

if (ql <= L && R <= qr)

{

tree[o].change += qans;

return;

}

push\_down(SELF);

int M = L + (R-L)/2;

if (ql <= M) insert(LSON);

else maintain(LSON);

if (qr > M) insert(RSON);

else maintain(RSON);

maintain(SELF);

}

其中maintain和push\_down操作也有所变化，因为要额外考虑版本号的信息

void push\_down(int o, int L, int R)//向下传递标记，但是不修改o节点val的信息

{

if (tree[o].change && L < R)

{

if (tree[lc].ver != qver)//确认儿子节点版本号是否一致

{

tree[++tail] = tree[lc];

tree[o].lson = tail;

tree[lc].ver = qver;

}

if (tree[rc].ver != qver)

{

tree[++tail] = tree[rc];

tree[o].rson = tail;

tree[rc].ver = qver;

}

tree[lc].change += tree[o].change;

tree[rc].change += tree[o].change;

tree[o].change = 0;

}

}

//更新o节点的val信息。那么就需要把儿子的信息汇总到o

//要特别注意，儿子的懒人标记的信息和val信息汇总

void maintain(int o, int L, int R) //没有涉及版本号的操作

{

if (L < R)

{

int M = L+(R-L)/2;

tree[o].val = tree[lc].val + tree[rc].val + tree[lc].change \* (M-L+1) + tree[rc].change \* (R-M);

}

//L==R的情况，不包含的左右儿子，所以再query里询问到的节点，直接结合处理了

}

(3)询问操作

询问操作也涉及到版本号的修改，所以要额外注意。

long long query(int o, int L, int R)

{

if (tree[o].ver != qver)

{

tree[++tail] = tree[o];

o = tail;

tree[o].ver = qver;

}

if (ql <= L && R <= qr)

{

//返回的值需要考虑到懒人标记信息

return tree[o].val + tree[o].change \* (R-L+1);//因为根节点的传递标记一直都在

}

push\_down(SELF);

int M = L + (R - L) / 2;

long long ret = 0;

if (ql <= M) ret += query(LSON);

else maintain(LSON);

if (qr > M) ret += query(RSON);

else maintain(RSON);

maintain(SELF);

return ret;

}

(4)主要函数

主要的函数，也就是这道题主的主函数。

void doit()

{

int now =0;

int que,l,r;

while (m--)

{

scanf("%s", inp);

if (inp[0] == 'C') //区间更新,now+1

{

++now;

root[now] = root[now - 1];

qver = now;

scanf("%d%d%d", &ql, &qr, &qans);//ql,qr区间变化qans

insert(root[now], 1, n);

}

if (inp[0]=='Q') //询问当前

{

scanf("%d%d", &ql, &qr);

qver = now;

printf("%lld\n",query(root[now], 1, n));

}

if (inp[0]=='H') //询问que时间的L,R区间和

{

scanf("%d%d%d", &ql, &qr, &qver);

printf("%lld\n",query(root[qver], 1, n));

}

if (inp[0] == 'B') //回到过去

{

scanf("%d", &now);

}

}

puts("");

}

## 莫比乌斯反演

（以后再填坑）

## C++ pb\_ds库的

1.需要头文件以及声明

#include <bits/stdc++.h>

#include<ext/pb\_ds/assoc\_container.hpp>

using namespace std;

using namespace \_\_gnu\_pbds;

1. 如何使用

首先我们先定义一个

typedef tree<int,null\_type,less<int>,rb\_tree\_tag,tree\_order\_statistics\_node\_update> shu;

这样我们可以声明一颗树了

shu tree; 这是一颗红黑树，定义中Tag可以更改，测试得到红黑树最快

Tag 有以下几种

rb\_tree\_tag

splay\_tree\_tag

ov\_tree\_tag (据说这叫排序向量树？)

1. 有哪些功能（有缺陷，会自动去重，后文补充支持可重）

注.以下操作如果没有特殊说明默认参数less

1. .PB\_DS的树支持插入和删除以及容器内的元素

tree.insert() , tree.erase() , tree.size()

(2).支持求第k小数 tree.find\_by\_order(k) 这样会返回一个第k+1 小的迭代器

所以要求值自然是 \*tree.find\_by\_order(k-1) 这样会得到一个第k小的数

(3).支持求第k大数 只需要修改定义中的参数less 修改为greater就可以得到第k大的

数（默认的是less参数）

1. . 求比x小的数有多少个（x的rank) tree.order\_of\_key(x) 返回值是rank -1 (rank从 0开始，所以如果要求的是rank结果就要+1)

附上4的代码

int x;

scanf("%d",&x);

printf("输出数%d的排名: %d\n",x,tree.order\_of\_key(x) + 1);

printf("输出比%d小的数有多少：%d\n",x,tree.order\_of\_key(x));

printf("输出第x小的数：%d\n",x,\*tree.find\_by\_order(x-1));

(5). 求比x大的数有多少个 同样的只要修改less为greater就可以了

1. 对于上述功能的补充支持可重集

必须自行搞一个偏移量。我的方法是把每个元素乘以2^20后加上偏移量，偏移量是‘当前重复了几次’，再用一个map来维护每个元素出现了几次。

功能和上述一样，代码实现如下

1. . 插入 tree.insert( (x<<20) + (MP[x]++) )
2. . 删除 tree.erase(T.find((x<<20)+(--MP[x]))); if(!MP[x]) MP.erase(MP.find(x)); 如果map里的元素都没了，才删除tree中的x数
3. . 询问

int x;

scanf("%d",&x);

printf("输出数%d的排名: %d\n",x,T.order\_of\_key( (x<<20) ) + 1);

printf("输出比%d小的数有多少：%d\n",x,T.order\_of\_key( (x<<20) ) );

printf("输出第x小的数：%d\n",x,\*T.find\_by\_order((x-1)>>20));

上述询问只需要把less改为greater 就可以支持相反操作了。

1. . 数x的前驱后继 直接在map上算

求x的前驱(前驱定义为小于x，且最大的数)

map<int,int>::iterator key=MP.lower\_bound(x);key--;

printf("%d\n",key->first);

求x的后继(后继定义为大于x，且最小的数)

map<int,int>::iterator key=MP.upper\_bound(x);

printf("%d\n",key->first);

(可以直接auto省事)

template <class Node\_CItr, class Node\_Itr,

class Cmp\_Fn, class \_Alloc>

struct my\_node\_update{

virtual Node\_CItr node\_begin() const = 0;

virtual Node\_CItr node\_end() const = 0;

typedef int metadata\_type;///这里都是固定格式 metadata\_type是指节点上记录的额外信息的类型

inline void operator()(Node\_Itr it,Node\_CItr end\_it)

{

Node\_Itr l = it.get\_l\_child(),r = it.get\_r\_child();

int left = 0,right = 0;

if(l!= end\_it) left = l.get\_metadata();

if(r!= end\_it) right = r.get\_metadata();

const\_cast<metadata\_type&>(it.get\_metadata())

= left + right + (\*it)->second;

} ///opetator()的功能是将节点it的信息更新为其左右孩子的信息之和，传入的end\_it表示空节点

inline int prefix\_sum(int x)

{

int ans = 0;

Node\_CItr it = node\_begin();

while(it!= node\_end())

{

Node\_CItr l = it.get\_l\_child(), r = it.get\_r\_child();

if(Cmp\_Fn()(x,(\*it)->first)) it = l;

else

{

ans += (\*it) ->second;

if(l!=node\_end()) ans += l.get\_metadata();

it = r;

}

}

return ans;

}

inline int interval\_sum(int l,int r)

{

return prefix\_sum(r) - prefix\_sum(l-1);

}

};

#include <bits/stdc++.h>

#include<ext/pb\_ds/assoc\_container.hpp>

#include<ext/pb\_ds/tree\_policy.hpp>

using namespace std;

using namespace \_\_gnu\_pbds;

///tree中存放结构体，为了比较我们需要自定义伪函数，所谓伪函数就是一个内部重载了()运算符的class，

///见代码中的Compare类。对于这道题，我们在Compare类中重载()运算符，实现“<”的功能，并将其放在上几题中greater的位置上。

///需要注意，Compare类中重载()运算符时，一定要在函数体前加一个const（见代码24行花括号前的const），代表该函数不会改变传入的元素，否则会编译失败。

template <class Node\_CItr, class Node\_Itr,

class Cmp\_Fn, class \_Alloc>

struct my\_node\_update{

virtual Node\_CItr node\_begin() const = 0;

virtual Node\_CItr node\_end() const = 0;

typedef int metadata\_type;///这里都是固定格式 metadata\_type是指节点上记录的额外信息的类型

inline void operator()(Node\_Itr it,Node\_CItr end\_it)

{

Node\_Itr l = it.get\_l\_child(),r = it.get\_r\_child();

int left = 1,right = 1;

if(l!= end\_it) left = l.get\_metadata();

if(r!= end\_it) right = r.get\_metadata();

const\_cast<metadata\_type&>(it.get\_metadata())

= left \* right \* (\*it)->second;

} ///opetator()的功能是将节点it的信息更新为其左右孩子的信息之和，传入的end\_it表示空节点

inline int prefix\_sum(int x,int m)

{

int ans = 1;

Node\_CItr it = node\_begin();

while(it!= node\_end())

{

Node\_CItr l = it.get\_l\_child(), r = it.get\_r\_child();

if(Cmp\_Fn()(x,(\*it)->first)) it = l;

else

{

ans =(ans%m)\*((\*it) ->second%m)%m;

if(l!=node\_end()) ans =(ans%m)\*(l.get\_metadata()%m)%m;

it = r;

}

}

return ans;

}

inline int interval\_sum(int l,int r)

{

return prefix\_sum(r) - prefix\_sum(l-1);

}

};

typedef tree<int,int,less<int>,rb\_tree\_tag,my\_node\_update> shu;///只需放在原来的上面就OK了

shu T;

int a[100000];

int b[100000];

void solve()

{

int t;

scanf("%d",&t);

while(t--)

{

int q,m;

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

int z = 0;

int i = 1;

int ii = 1;

T[0] = 1;

while(q--)

{

int x,y;

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

a[i++] = y;

//b[ii++] = x;

if(x==1)

{

T[++z] = y;

printf("%d\n",T.prefix\_sum(z,m));

}

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

if(x==2)

{

int ii;

for( ii = 1;ii<=q;ii++)

{

if(T[ii]==a[y])

{

break;

}

}

T[ii] = 1;

printf("%d\n",T.prefix\_sum(z,m));

}

}

}

}

int main()

{

solve();

return 0;

}

# 几何模板：

平面计算几何

---------

## //##1##:点的定义

//需要条件：无

1. const long double PI = acos(-1.0);
2. const double eps = 1e-8;
3. inline int dcmp(double x) { return (x>eps) - (x<-eps); }
4. //#####定义#####
5. #define Vector Point
6. struct Point {
7. double x, y; //int belong;//属于哪一个圆
8. Point(double x = 0, double y = 0) : x(x), y(y) {}
9. Vector operator + (const Vector& rhs) const {
10. return Vector(x + rhs.x, y + rhs.y); }
11. Vector operator - (const Point& rhs) const {
12. return Vector(x - rhs.x, y - rhs.y); }
13. Vector operator \* (double p) const {
14. return Vector(x \* p, y \* p); }
15. Vector operator / (double p) const {
16. return Vector(x / p, y / p); }
17. bool operator < (const Point& rhs) const {
18. return dcmp(x - rhs.x) < 0
19. || (dcmp(x-rhs.x)==0 && dcmp(y-rhs.y) < 0); }
20. // bool operator < (const Point& rhs) const {
21. //return x < rhs.x || (x == rhs.x && y < rhs.y); }
22. bool operator == (const Point& rhs) const {
23. return dcmp(x - rhs.x) == 0 && dcmp(y - rhs.y) == 0; }
24. bool operator > (const Point& rhs) const {
25. return !(\*this < rhs || \*this == rhs); }
26. bool operator >= (const Point& rhs) const {
27. return !(\*this < rhs); }
28. bool operator <= (const Point& rhs) const {
29. return (\*this < rhs || \*this == rhs); }
30. double operator \* (const Vector& rhs) const {
31. return x \* rhs.x + y \* rhs.y; } //点积
32. double operator ^ (const Vector& rhs) const {
33. return x \* rhs.y - y \* rhs.x; } //叉积
34. Point in() { scanf("%lf%lf", &x, &y); return \*this; }
35. void out() const { printf("(%lf, %lf) ", x, y); }
36. };

## //##2##:向量的基本运算

//需要条件：1

1. //向量的模
2. double length(Vector A) { return sqrt(A \* A); }
3. //向量的夹角 a\*b=|a||b|cosα
4. double angle(Vector A, Vector B) {
5. return acos(A \* B / length(A) / length(B)); }
6. //AB x AC组成的空间
7. double area2(Point A, Point B, Point C) {
8. return (B - A) ^ (C - A); }
9. //向量逆时针旋转rad弧度
10. Vector rotate(Vector A, double rad) {
11. return Vector(A.x\*cos(rad)-A.y\*sin(rad),
12. A.x\*sin(rad)+A.y\*cos(rad));
13. }
14. //计算A的法线（A不是0向量）
15. //左转90° 长度归一化
16. Vector normal(Vector A) {
17. double L = length(A);
18. return Vector(-A.y / L, A.x / L);
19. }
20. //计算向量极角(need<cmath>)
21. double angle(Vector v) { return atan2(v.y, v.x); }

## //##3##:两直线交点（没有不能用）(P+tv & Q+tw，后面有Line, Line)

//需要条件：1

1. Point get\_line\_intersection(Point P, Vector v, Point Q, Vector w) {
2. Vector u = P - Q;
3. double t = (w ^ u) / (v ^ w);
4. return P + v \* t;
5. }

## //##4##:P点到直线AB的距离

//需要条件：1, 2

1. double distance\_2\_line(Point P, Point A, Point B) {
2. Vector v1 = B - A, v2 = P - A;
3. return fabs(v1 ^ v2) / length(v1);
4. }

## //##5##:P点到线段AB的距离

//需要条件：1, 2

1. double distance\_2\_segment(Point P, Point A, Point B) {
2. if (A == B) return length(P - A);
3. Vector v1 = B - A, v2 = P - A, v3 = P - B;
4. if (dcmp(v1 \* v2) < 0) return length(v2);
5. else if (dcmp(v1 \* v3) > 0) return length(v3);
6. else return fabs(v1 ^ v2) / length(v1);
7. }

## //##6##:P点在直线AB上的投影

//需要条件：1

1. Point get\_line\_projection(Point P, Point A, Point B) {
2. Vector v = B - A;
3. return A + v \* (v \* (P - A) / (v \* v));
4. }

## //##7##:判断点P是否在线段a1a2上(不含端点)

//需要条件：1

1. bool on\_segment(Point P, Point a1, Point a2) {
2. return dcmp((a1-P) ^ (a2-P))==0 && dcmp((a1-P) \* (a2-P))<0;
3. }

## //##8##:线段a1a2,b1b2相交判定(线段无端点)

//如果要判断端点，用on\_segment

//需要条件：1

1. bool segment\_proper\_intersection(Point a1, Point a2,
2. Point b1, Point b2) {
3. double c1 = (a2 - a1) ^ (b1 - a1), c2 = (a2 - a1) ^ (b2 - a1),
4. c3 = (b2 - b1) ^ (a1 - b1), c4 = (b2 - b1) ^ (a2 - b1);
5. return dcmp(c1) \* dcmp(c2) < 0 && dcmp(c3) \* dcmp(c4) < 0;
6. }

## //##9##:点的极角排序

//需要条件：1, 2

1. bool angle\_cmp(const Point& a, const Point& b)
2. {
3. // return atan2(a.y, a.x) < atan2(b.y, b.x);
4. if (a.y \* b.y > 0) { //同上同下
5. if ((a ^ b) != 0) return (a ^ b) > 0;
6. return length(a) > length(b); //极角相同 距离原点远的小
7. }
8. if (a.y \* b.y < 0) return a.y < 0;//一上一下
9. if (a.y == 0) {
10. if (b.y == 0) {
11. if (a.x \* b.x < 0) return a.x > b.x;
12. return length(a) > length(b); //极角相同
13. }
14. if (a.x > 0) return b.y > 0;
15. return false;
16. }
17. if (b.x > 0) return a.y < 0;
18. return true;
19. }

## //##10##:有向直线。他的左边是对应的半平面

//需要条件：1, 2

1. struct Line {
2. Point p; //直线上的任意一点(线段的时候可当作起点
3. Vector v; //方向向量
4. //double ang;//极角
5. Line() {}
6. Line(Point p, Vector v):p(p),v(v) { /\*ang = atan2(v.y, v.x);\*/}
7. Point point(double t) { return p + v\*t; }
8. Line move(double d) { return Line(p + normal(v)\*d, v); }
9. bool operator < (const Line& rhs) const {
10. //根据原点极角排序(小的在前，极角相同 长度小的在前)
11. //return ang < L.ang;
12. if (v.y \* rhs.v.y > 0) { //同上同下
13. if ((v ^ rhs.v) != 0) return (v ^ rhs.v) > 0;
14. return length(v) > length(rhs.v); //极角相同 长的小
15. }
16. if (v.y \* rhs.v.y < 0) return v.y < 0;//一上一下
17. if (v.y == 0) {
18. if (rhs.v.y == 0) {
19. if (v.x \* rhs.v.x < 0) return v.x > rhs.v.x;
20. return length(v) > length(rhs.v); //极角相同长的小
21. }
22. if (v.x > 0) return rhs.v.y > 0;
23. return false;
24. }
25. if (rhs.v.x > 0) return v.y < 0;
26. return true;
27. }
28. };

## //##11##:二直线交点，假定交点惟一存在

//需要条件：1 2 10;

1. Point get\_line\_intersection(const Line& a, const Line& b) {
2. Vector u = a.p-b.p;
3. double t = (b.v ^ u) / (a.v ^ b.v);
4. return a.p + a.v\*t;
5. }

## //##12##:圆的定义

//需要条件：1

1. struct Circle {
2. Point c;
3. double r;
4. Circle(Point c, double r):c(c), r(r) {}
5. Circle(){}
6. Point point(double a) {//给极角求坐标
7. return Point(c.x + cos(a)\*r, c.y + sin(a)\*r);
8. }
9. };

## //##13##:判断点在圆心内。圆周上不算

//需要条件：1, 12

1. bool is\_in\_circle(Point p, Point center, double R) {
2. return dcmp((p-center)\*(p-center) - R\*R) < 0;
3. }

## //##14##:直线与圆的交点

//需要条件：1 2 10 12

1. int get\_line\_circle\_intersection(Line L, Circle C,
2. double& t1, double& t2, vector<Point>& sol) {
3. //t1,t2为从L.p出发的长度，sol保存结果，返回值为交点个数
4. double a = L.v.x, b = L.p.x - C.c.x,
5. c = L.v.y, d = L.p.y - C.c.y;
6. double e = a\*a + c\*c, f = 2\*(a\*b + c\*d),
7. g = b\*b + d\*d - C.r\*C.r;
8. double delta = f\*f - 4\*e\*g; // 判别式
9. if (dcmp(delta) < 0) return 0; // 相离
10. if (dcmp(delta) == 0) { // 相切
11. t1 = t2 = -f / (2 \* e);
12. sol.push\_back(L.point(t1));
13. return 1;
14. }
15. // 相交
16. t1 = (-f - sqrt(delta)) / (2 \* e); sol.push\_back(L.point(t1));
17. t2 = (-f + sqrt(delta)) / (2 \* e); sol.push\_back(L.point(t2));
18. return 2;
19. }

## //##15##:圆和线段是否相交（相切不算）。线段不考虑端点

//需要条件：1 2 10 12 14

1. bool circle\_intersect\_segment(Point A, Point B, Point p,
2. double R) {
3. double t1, t2;
4. vector<Point> sol; //直线A B与圆的交点
5. int c = get\_line\_circle\_intersection(Line(A, B-A),
6. Circle(p, R), t1, t2, sol);
7. if(c <= 1) return false;
8. if(dcmp(t1) > 0 && dcmp(t1-1) < 0) return true; // 端点在圆上
9. if(dcmp(t2) > 0 && dcmp(t2-1) < 0) return true;
10. return false;
11. }

## //##16##:两圆相交的交点

//需要条件：1 2 12

1. int get\_circle\_circle\_intersection(Circle C1, Circle C2,
2. vector<Point>& sol) {
3. //sol保存结果，返回值为交点个数
4. double d = length(C1.c - C2.c);
5. if (dcmp(d) == 0) {
6. if (dcmp(C1.r - C2.r) == 0) return -1; // 重合，无穷多交点
7. return 0;
8. }
9. if (dcmp(C1.r + C2.r - d) < 0) return 0;
10. if (dcmp(fabs(C1.r-C2.r) - d) > 0) return 0;
11. double a = angle(C2.c - C1.c);
12. double da = acos((C1.r\*C1.r + d\*d - C2.r\*C2.r) / (2\*C1.r\*d));
13. Point p1 = C1.point(a-da), p2 = C1.point(a+da);
14. sol.push\_back(p1);
15. if (p1 == p2) return 1;
16. sol.push\_back(p2);
17. return 2;
18. }

## //##17##:把角变成0~2pi范围内(负数也可以用

//需要条件: 无

1. double normalize\_angle(double rad) {
2. return rad - 2 \* PI \* floor(rad / (2 \* PI));
3. }

## //##18##:两圆相交的交点相对于圆1的极角保存在rad中

//需要条件: 1 2 12 17

1. void get\_circle\_circle\_intersection(Circle C1, Circle C2,
2. vector<double>& rad) {
3. double d = length(C1.c - C2.c);
4. if(dcmp(d) == 0) return; // 不管是内含还是重合，都不相交
5. if(dcmp(C1.r + C2.r - d) < 0) return;
6. if(dcmp(fabs(C1.r-C2.r) - d) > 0) return;
7. double a = angle(C2.c - C1.c);
8. double da = acos((C1.r\*C1.r + d\*d - C2.r\*C2.r) / (2\*C1.r\*d));
9. rad.push\_back(normalize\_angle(a-da));//相交弧为从减到加（逆时针
10. rad.push\_back(normalize\_angle(a+da));
11. }

## //##19##:过点p到圆C的切线。v[i]是第i条切线的向量。返回切线条数

//需要条件: 1 2 12

1. int get\_tangents(Point p, Circle C, Vector\* v) {
2. Vector u = C.c - p;
3. double dist = length(u);
4. if (dist < C.r) return 0;
5. if (dcmp(dist - C.r) == 0) { // p在圆上，只有一条切线
6. v[0] = rotate(u, PI/2);
7. return 1;
8. }
9. double ang = asin(C.r / dist);
10. v[0] = rotate(u, -ang);
11. v[1] = rotate(u, +ang);
12. return 2;
13. }

## //##20##:过点p到圆C的切线。

//V[i]是第i条切线的向量。P[i]为切点 返回切线条数

//需要条件: 1 2 12

1. int get\_tangents(Point p, Circle C, vector<Vector>& V,
2. vector<Point>& P) {
3. Point u = p - C.c;
4. double dist = length(u);
5. if (dcmp(dist - C.r) < 0) return 0;
6. if (dcmp(dist - C.r) == 0) {//多边形的点在圆上要改成切点
7. V.push\_back(rotate(u, PI/2));
8. P.push\_back(p);
9. return 1;
10. }
11. double ang = acos(C.r / dist);
12. V.push\_back(rotate(u, -ang));
13. P.push\_back(C.c + V[V.size()-1] / length(V[V.size()-1]) \* C.r);
14. V.push\_back(rotate(u, ang));
15. P.push\_back(C.c + V[V.size()-1] / length(V[V.size()-1]) \* C.r);
16. return 2;
17. }

## //##21##:过点p到圆C的切线 P为切点

//需要条件: 1 2 12 19

//!!!!待验证

1. int get\_tangents\_point(Point p, Circle C, vector<Point>& P) {
2. Vector v[3];
3. int num = get\_tangents(p, C, v);
4. if (num == 0) return 0;
5. if (num == 1) {
6. P.push\_back(p);
7. return 1;
8. }
9. P.push\_back(C.point(atan2(p.y, p.x)
10. + PI/2-angle((C.c-p), v[0])));
11. P.push\_back(C.point(atan2(p.y, p.x)
12. - (PI/2-angle((C.c-p), v[0]))));
13. return 2;
14. }

## //##22##:两圆的公切线，返回切线条数，-1为无穷

//需要条件: 1 12

1. int get\_tangents2(Circle A, Circle B, Point\* a, Point\* b) {
2. //a[i]和b[i]分别是第i条切线在圆A和圆B上的切点
3. int cnt = 0;
4. if (A.r < B.r) { swap(A, B); swap(a, b); }
5. int d2 = (A.c.x-B.c.x)\*(A.c.x-B.c.x)
6. + (A.c.y-B.c.y)\*(A.c.y-B.c.y);
7. int rdiff = A.r-B.r;
8. int rsum = A.r+B.r;
9. if (d2 < rdiff \* rdiff) return 0; //内含
10. double base = atan2(B.c.y-A.c.y, B.c.x-A.c.x);
11. if (d2 == 0 && A.r == B.r) return -1;
12. if (d2 == rdiff \* rdiff) {
13. a[cnt] = A.point(base); b[cnt] = B.point(base);
14. ++cnt; return 1;
15. }
16. double ang = acos((A.r-B.r) / sqrt(d2));
17. a[cnt] = A.point(base+ang);
18. b[cnt++] = B.point(base+ang);
19. a[cnt] = A.point(base-ang);
20. b[cnt++] = B.point(base-ang);
21. if (d2 == rsum \* rsum) {
22. a[cnt] = A.point(base); b[cnt] = B.point(PI+base); ++cnt;
23. }
24. else if (d2 > rsum \* rsum) {
25. double ang = acos((A.r+B.r) / sqrt(d2));
26. a[cnt] = A.point(base+ang);
27. b[cnt++] = B.point(PI+base+ang);
28. a[cnt] = A.point(base-ang);
29. b[cnt++] = B.point(PI+base-ang);
30. }
31. return cnt;
32. }

## //##23##:三角形外接圆

//需要条件: 1 2 12

1. Circle circumscribed\_circle(Point p1, Point p2, Point p3) {
2. double Bx = p2.x-p1.x, By = p2.y-p1.y;
3. double Cx = p3.x-p1.x, Cy = p3.y-p1.y;
4. double D = 2\*(Bx\*Cy-By\*Cx);
5. double cx = (Cy\*(Bx\*Bx+By\*By) - By\*(Cx\*Cx+Cy\*Cy))/D + p1.x;
6. double cy = (Bx\*(Cx\*Cx+Cy\*Cy) - Cx\*(Bx\*Bx+By\*By))/D + p1.y;
7. Point p = Point(cx, cy);
8. return Circle(p, length(p1-p));
9. }

## //##24##:三角形内切圆

//需要条件: 1 2 4 12

1. Circle inscribed\_circle(Point p1, Point p2, Point p3) {
2. double a = length(p2-p3);
3. double b = length(p3-p1);
4. double c = length(p1-p2);
5. Point p = (p1\*a+p2\*b+p3\*c)/(a+b+c);
6. return Circle(p, distance\_2\_line(p, p1, p2));
7. }

## //##25##:经过点p且与直线L相切，半径为r的圆，返回值为圆心

//需要条件: 1 2 10 12 14

1. vector<Point> circle\_through\_point\_tangent2line\_given\_radius
2. (Point p, Line L, double r) {
3. vector<Point> ans;
4. double t1, t2;
5. get\_line\_circle\_intersection(L.move(-r),
6. Circle(p, r), t1, t2, ans);
7. get\_line\_circle\_intersection(L.move(r),
8. Circle(p, r), t1, t2, ans);
9. return ans;
10. }

## //##26##: 半径为r且与两条直线同时相切的圆 返回值为圆心

//需要条件: 1 2 10 11

1. vector<Point> circle\_tangent2lines\_given\_radius(Line a, Line b,
2. double r) {
3. vector<Point> ans;
4. Line L1 = a.move(-r), L2 = a.move(r);
5. Line L3 = b.move(-r), L4 = b.move(r);
6. ans.push\_back(get\_line\_intersection(L1, L3));
7. ans.push\_back(get\_line\_intersection(L1, L4));
8. ans.push\_back(get\_line\_intersection(L2, L3));
9. ans.push\_back(get\_line\_intersection(L2, L4));
10. return ans;
11. }

## //##27##: 求与两个圆外切的半径为r的圆, 返回圆心

//需要条件: 1 2 12 16

1. vector<Point> circle\_tangent2two\_disjoint\_circles\_with\_radius
2. (Circle c1, Circle c2, double r) {
3. vector<Point> ans;
4. Vector v = c2.c - c1.c;
5. double dist = length(v);
6. int d = dcmp(dist - c1.r -c2.r - r\*2);//两圆相距太远
7. if(d > 0) return ans;
8. get\_circle\_circle\_intersection(Circle(c1.c, c1.r+r),
9. Circle(c2.c, c2.r+r), ans);
10. return ans;
11. }

## //##28##: 线段和圆的交点 结果保存在sol中

//需要条件: 1 2 10 12 14

1. int get\_segment\_circle\_intersection(Point p1, Point p2,
2. Circle C, vector<Point>& sol) {
3. double t1, t2;
4. vector<Point> sol\_t;
5. int m = get\_line\_circle\_intersection(Line(p1, p2 - p1),
6. C, t1, t2, sol\_t); if (m == 0) return 0;
7. if (m <= 2) {
8. if (dcmp((p1 - sol\_t[0]) \* (p2 - sol\_t[0])) <= 0) {
9. sol.push\_back(sol\_t[0]);
10. }
11. if (m == 1) return sol.size();
12. }
13. if (dcmp((p1 - sol\_t[1]) \* (p2 - sol\_t[1])) <= 0) {
14. sol.push\_back(sol\_t[1]);
15. }
16. return sol.size();
17. }

## //##29##: 多边形定义

//需要条件: 1

1. typedef vector<Point> Polygon; //组成该平面的逆时针点集

## //##30##: 多边形的有向面积(点逆时针旋转为正向)(顺时针将求出负值)

//需要条件: 1 29

1. double polygon\_area(const Polygon& P) {
2. if (P.size() == 0) return 0;
3. double area = 0;
4. for (int i = 1; i < P.size() - 1; ++i)
5. area += ((P[i] - P[0]) ^ (P[i + 1] - P[0]));
6. return area / 2;
7. }

## //##31##: 点在多边形内判定（不是简单多边形 有弧边也可以使用）

//需要条件: 1 7 29

1. int is\_point\_in\_polygon(Point p, const Polygon& poly) {
2. int wn = 0;
3. int n = poly.size();
4. for (int i = 0; i < n; i++){
5. if (on\_segment(p, poly[i], poly[(i+1)%n])) return -1;//边界上
6. int k = dcmp((poly[(i+1)%n]-poly[i]) ^ (p-poly[i])); //叉积
7. int d1 = dcmp(poly[i].y - p.y);
8. int d2 = dcmp(poly[(i+1)%n].y - p.y);
9. if (k > 0 && d1 <= 0 && d2 > 0) wn++;
10. if (k < 0 && d2 <= 0 && d1 > 0) wn--;
11. }
12. if (wn != 0) return 1; // 内部
13. return 0; // 外部
14. }

## //##32##: 多边形重心

//需要条件: 1 29

1. Point get\_polygon\_center\_of\_gravity(Polygon poly) {
2. int n = poly.size();
3. Point ret;
4. double area = 0, area2;
5. for (int i = 1; i < n; i++) {
6. area2 = (poly[i-1] ^ poly[i]); //上下都取面积的2倍, 被约掉
7. ret = ret + (poly[i-1] + poly[i]) \* area2;
8. area += area2;
9. }
10. area2 = (poly[n-1] ^ poly[0]);
11. ret = ret + (poly[n-1] + poly[0]) \* area2;
12. area += area2;
13. return ret / (area \* 3);
14. }

## //##33##: 圆和凸多边形相交的面积

//圆心必须在凸多边形内部（不含边

//需要条件: 1 2 10 12 14 28 29

1. double area\_circle\_intersection\_polygon(Circle C, Polygon& P) {
2. P.push\_back(P[0]);
3. double ret = 0;
4. for (int i = 0; i < P.size() - 1; i++) {
5. bool flag1 = (dcmp(length(C.c - P[i]) - C.r) <= 0);
6. bool flag2 = (dcmp(length(C.c - P[i+1]) - C.r) <= 0);
7. if (flag1 + flag2 == 2) { //两个点都在圆内 算三角形
8. ret += fabs((P[i] - C.c) ^ (P[i+1] - C.c)) / 2;
9. continue;
10. }
11. vector<Point> sol;
12. int num = get\_segment\_circle\_intersection(P[i], P[i+1],
13. C, sol);
14. if (flag1 + flag2 == 1) {//一个点在圆内一个点在圆外
15. if (flag1) {
16. ret += C.r\*C.r\*angle(P[i+1]-C.c, sol[0]-C.c)/2
17. + fabs((P[i]-C.c)^(sol[0]-C.c))/2;
18. continue;
19. }
20. ret += C.r\*C.r\*angle(P[i]-C.c, sol[0]-C.c)/2
21. + fabs((P[i+1]-C.c)^(sol[0]-C.c))/2;
22. continue;
23. }
24. //两个点都在圆外
25. if (num == 2) {
26. ret += C.r\*C.r\*(angle(P[i]-C.c, sol[0]-C.c)
27. + angle(P[i+1]-C.c, sol[1]-C.c))/2
28. + fabs((sol[0]-C.c)^(sol[1]-C.c)) / 2;
29. continue;
30. }
31. ret += C.r\*C.r\*angle(P[i]-C.c, P[i+1]-C.c)/2;
32. }
33. P.pop\_back();
34. return ret;
35. }

## //##34##: 返回圆盘是否与面poly相交

//需要条件: 1 2 7 10 12 13 14 15 29 31

1. bool disc\_intersect\_polygon(Polygon poly, Point p, double R) {
2. if(is\_point\_in\_polygon(p, poly) != 0) return true;
3. if(is\_in\_circle(poly[0], p, R)) return true;
4. int n = poly.size();
5. for(int i = 0; i < n; i++) {
6. if(circle\_intersect\_segment(poly[i], poly[(i+1)%n], p, R)) {
7. return true; // 不考虑线段端点
8. }
9. if(is\_in\_circle((poly[i]+poly[(i+1)%n])\*0.5, p, R)) {
10. return true; // 两个端点都在圆上
11. }
12. }
13. return false;
14. }

## //##35##: 删除平面的三点共线

//假定poly没有相邻点重合的情况，只需要删除三点共线的情况

//需要条件: 1 29

1. Polygon simplify(const Polygon& poly) {
2. Polygon ans;
3. int n = poly.size();
4. //测试每一个点是否和前后两点共线，如果不是，就加入
5. for(int i = 0; i < n; i++) {
6. Point a = poly[i];
7. Point b = poly[(i+1)%n];
8. Point c = poly[(i+2)%n];
9. if(dcmp((a-b) ^ (c-b)) != 0) ans.push\_back(b);
10. }
11. return ans;
12. }

## //##36##: 用有向直线AB切割poly 返回切割后的左边

//可能会返回一个点或一条线段

//需要条件: 1 3 7 29

1. Polygon cut\_polygon(const Polygon& poly, Point A, Point B) {
2. Polygon newpoly;
3. int n = poly.size();
4. for(int i = 0; i < n; i++) {
5. Point C = poly[i];
6. Point D = poly[(i+1)%n];
7. if(dcmp((B-A) ^ (C-A)) >= 0) newpoly.push\_back(C);
8. if(dcmp((B-A) ^ (C-D)) != 0) {
9. Point ip = get\_line\_intersection(A, B-A, C, D-C);
10. if(on\_segment(ip, C, D)) newpoly.push\_back(ip);
11. }
12. }
13. return newpoly;
14. }

//##37##: Andrew算法，点集凸包

// 如果不希望在凸包的边上有输入点，把两个 <= 改成 <

// 如果不介意点集被修改，可以改成传递引用(不能修改就去掉&

// 保证P非空

//需要条件: 1

1. vector<Point> convex\_hull(vector<Point>& P) {
2. // 预处理，删除重复点
3. sort(P.begin(), P.end());//两个点比大小，先横坐标再纵坐标，升序
4. int n = unique(P.begin(), P.end()) - P.begin();//删除重复点
5. int m = 0;
6. vector<Point> ch(n+1);
7. for (int i = 0; i < n; i++) {//找到下凸包
8. while (m > 1 && ((ch[m-1]-ch[m-2]) ^ (P[i]-ch[m-2])) <= 0)
9. m--;
10. ch[m++] = P[i];//发现右边的点时删除前面的点，再更新
11. }
12. int k = m;//下凸包点数，找上凸包时防止误删
13. for (int i = n-2; i >= 0; i--) {
14. while (m > k && ((ch[m-1]-ch[m-2]) ^ (P[i]-ch[m-2])) <= 0)
15. m--;
16. ch[m++] = P[i];
17. }
18. if (n > 1) m--;
19. ch.resize(m);
20. return ch;//返回的点集逆时针排序
21. }

## //##38##: 半平面交

//需要条件: 1 2 10 11 29

1. // 点p在有向直线L的左边（线上不算）
2. bool on\_left(const Line& L, const Point& p) {
3. return (L.v ^ (p-L.p)) > 0; }
4. // 如果不介意边被修改 可以改为传引用
5. Polygon halfplane\_intersection(vector<Line> L) {
6. int n = L.size();
7. sort(L.begin(), L.end()); // 按极角排序
8. int first, last; // 双端队列的第一个元素和最后一个元素的下标
9. vector<Point> p(n); // p[i]为q[i]和q[i+1]的交点
10. vector<Line> q(n); // 双端队列
11. Polygon ans; // 结果
12. q[first=last=0] = L[0]; // 双端队列初始化为只有一个半平面L[0]
13. for (int i = 1; i < n; i++) {
14. while (first < last && !on\_left(L[i], p[last-1])) last--;
15. while (first < last && !on\_left(L[i], p[first])) first++;
16. q[++last] = L[i];
17. if (fabs(q[last].v ^ q[last-1].v) < eps) {
18. //两向量平行且同向 取内侧一个
19. last--;
20. if (on\_left(q[last], L[i].p)) q[last] = L[i];
21. }
22. if (first<last)
23. p[last-1] = get\_line\_intersection(q[last-1], q[last]);
24. }
25. while (first<last && !on\_left(q[first], p[last-1]))
26. last--;//删除无用平面
27. if (last - first <= 1) return ans; // 空集
28. //计算首尾两个半平面的交点
29. p[last] = get\_line\_intersection(q[last], q[first]);
30. // 从deque复制到输出中
31. for (int i = first; i <= last; i++) ans.push\_back(p[i]);
32. return ans;
33. }

## //##39##: 旋转卡壳求点集直径的平方

//需要条件: 1 37

1. //两点距离的平方
2. double dist2(const Point& A, const Point& B) {
3. return (A.x-B.x)\*(A.x-B.x) + (A.y-B.y)\*(A.y-B.y); }
4. double diameter2(vector<Point>& pt) {
5. vector<Point> p = convex\_hull(pt);
6. int n = p.size();
7. if (n == 1) return 0;
8. if (n == 2) return dist2(p[0], p[1]);
9. p.push\_back(p[0]); //免得取模
10. double ans = 0;
11. for (int u = 0, v = 1; u < n; u++) {
12. // 一条直线贴住边p[u]-p[u+1]
13. for (;;) { //因为是平行直线 如果距离最远 则面积最大
14. // 当Area(p[u], p[u+1], p[v+1])
15. // <= Area(p[u], p[u+1], p[v])时停止旋转
16. // 即(p[u+1]-p[u]) ^ (p[v+1]-p[u])
17. // - (p[u+1]-p[u]) ^ (p[v]-p[u]) <= 0
18. // 根据(A^B)-(A^C) = (A^(B-C))
19. // 化简得(p[u+1]-p[u]) ^ (p[v+1]-p[v]) <= 0
20. int diff = dcmp((p[u+1]-p[u]) ^ (p[v+1]-p[v]));
21. if(diff <= 0) {
22. ans = max(ans, dist2(p[u], p[v]));//u和v是对踵点
23. if(diff == 0) ans = max(ans, dist2(p[u], p[v+1]));
24. //diff == 0时u和v+1也是对踵点
25. break;
26. }
27. v = (v + 1) % n;
28. }
29. }
30. return ans;
31. }

## //##40##: 旋转卡壳求能覆盖点集poly的 面积和周长最小的矩形

//S为面积，C为周长

//需要条件: 1 2 37

1. void rotating\_calipers(vector<Point>& poly, double& S, double& C) {
2. vector<Point> P(convex\_hull(poly));
3. int t = 1, L = 1, R = 1, n = P.size();
4. S = C = LINF;
5. P.push\_back(P[0]);
6. for (int i = 0; i < n; i++) {//以P[i+1] - P[i]为底部
7. while (dcmp((P[i+1]-P[i])^(P[t+1]-P[t])) > 0)
8. t = (t + 1) % n; //上边
9. while (dcmp((P[i+1]-P[i])\*(P[R+1]-P[R])) > 0)
10. R = (R + 1) % n; //右边
11. if (i == 0) L = (R + 1) % n; //初始化最左点
12. while (dcmp((P[i+1]-P[i])\*(P[L+1]-P[L])) < 0)
13. L = (L + 1) % n; //左边
14. double d = length(P[i] - P[i+1]);
15. //平行四边形面积/底=高
16. double h = ((P[i+1] - P[i]) ^ (P[t] - P[i])) / d;
17. double w = ((P[i+1] - P[i]) \* (P[R] - P[L])) / d; //投影
18. S = min(S, h \* w);
19. C = min(C, 2 \* (h + w));
20. }
21. }

## //##41##: 平面直线图

//需要条件: 1 29 30

//调用方法:给出点和边，求面

//init(n) n为点数

//add\_point(const vector<Point>& V) 传入点集

//add\_edge(u, v) 添加u到v的变 u,v为点的序号

//get\_face(),得到如下东西

//faces 保存每个面（面是由逆时针点构成）

//left[i] 每条边 左边的面的编号

//area[i] 每个面的面积

//对于内部区域来说，无限面的各个顶点是顺时针的

//无限面多边形上可能会有相邻共线点

//对于连通图，唯一一个面积小于0的面是无限面

1. struct Edge { int u, v; /\*double ang;\*/};
2. const int maxn = 10000 + 10;
3. //平面直线图
4. static double \*xp, \*yp;
5. static vector<Edge>\* ep;
6. struct PSLG {
7. int n, m, face\_cnt;
8. double x[maxn], y[maxn];
9. vector<Edge> edges;
10. vector<int> G[maxn];
11. int vis[maxn\*2]; //每条边是否已经访问过
12. int left[maxn\*2];//左边的编号
13. int prev[maxn\*2];//相同起点的上一条边（右边的边）的变化
14. vector<Polygon> faces;
15. double area[maxn]; //每个polygon的面积
16. void init(int n) {
17. this->n = n;
18. for (int i = 0; i < n; i++) G[i].clear();
19. edges.clear(); faces.clear();
20. xp = x; yp = y; ep = &edges;
21. }
22. /\*double get\_angle(int u, int v)
23. { return atan2(y[v]-y[u], x[v]-x[u]); }\*/
24. void add\_point(const vector<Point>& V) {
25. for (int i = 0; i < n; i++) {
26. x[i] = V[i].x;
27. y[i] = V[i].y;
28. }
29. }
30. void add\_edge(int u, int v) {
31. edges.push\_back((Edge){u, v/\*, get\_angle(u, v)\*/});
32. edges.push\_back((Edge){v, u/\*, get\_angle(v, u)\*/});
33. m = edges.size();
34. G[u].push\_back(m-2);
35. G[v].push\_back(m-1);
36. }
37. static bool cmp(int i, int j) {//极角排序 辅助排序
38. //return (\*ep)[i].ang < (\*ep)[j].ang;
39. double x1 = xp[(\*ep)[i].v] - xp[(\*ep)[i].u];
40. double x2 = xp[(\*ep)[j].v] - xp[(\*ep)[j].u];
41. double y1 = yp[(\*ep)[i].v] - yp[(\*ep)[i].u];
42. double y2 = yp[(\*ep)[j].v] - yp[(\*ep)[j].u];
43. if (y1 \* y2 > 0)
44. return (Vector(x1, y1) ^ Vector(x2, y2)) > 0;
45. if (y1 \* y2 < 0) return y1 < 0;
46. if (y1 == 0 && y2 == 0) return x1 > x2;
47. if (y1 == 0 && x1 > 0) return y2 > 0;
48. if (y1 == 0 && x1 <= 0) return false;
49. if (x2 > 0) return y1 < 0;
50. return true;
51. }
52. //找出faces, 并计算面积
53. void get\_faces() {
54. for (int u = 0; u < n; u++) {
55. //给从u出发的各条边按极角排序
56. sort(G[u].begin(), G[u].end(), cmp);
57. int d = G[u].size();
58. /\*for(int i = 0; i < d; i++)
59. for(int j = i+1; j < d; j++)
60. if(edges[G[u][i]].ang > edges[G[u][j]].ang)
61. swap(G[u][i], G[u][j]);\*/
62. for (int i = 0; i < d; i++)
63. prev[G[u][(i+1)%d]] = G[u][i];
64. }
65. memset(vis, 0, sizeof(vis));
66. face\_cnt = 0;
67. for (int u = 0; u < n; u++) {
68. for (int i = 0; i < G[u].size(); i++) {
69. int e = G[u][i];
70. if (!vis[e]) { // 逆时针找圈
71. face\_cnt++;
72. Polygon poly;
73. for (;;) {
74. vis[e] = 1; left[e] = face\_cnt;
75. int from = edges[e].u;
76. poly.push\_back(Point(x[from], y[from]));
77. e = prev[e^1];
78. if (e == G[u][i]) break;
79. }
80. faces.push\_back(poly);
81. }
82. }
83. }
84. for(int i = 0; i < faces.size(); i++) {
85. area[i] = polygon\_area(faces[i]);
86. }
87. }
88. } g;

## 其他几何操作

### 经纬度求球面最短距离

1. //lon是经度 lat是纬度
2. double sphereDis(double lon1, double lat1, double lon2, double lat2, double R) {
3. return R \* acos(cos(lat1) \* cos(lat2) \* cos(lon1 - lon2) + sin(lat1) \* sin(lat2));
4. }

### 三角形的心

1. //传入的参数 point a,b,c; 三角形顶点
2. double area(point a,point b,point c) { return(fabs(det(b-a,c-a))/2); }// 面积
3. point barycenter(point a,point b,point c) // 重心
4. { return(point((a.x+b.x+c.x)/3.0,(a.y+b.y+c.y)/3.0)); }
5. point orthocenter(point a,point b,point c) // 垂心
6. { double dx,dy,d=(c.x-b.x)\*(c.y-a.y)-(c.x-a.x)\*(c.y-b.y);
7. dx=(a.y\*(c.y-b.y)+a.x\*(c.x-b.x))\*(c.y-a.y)-(b.y\*(c.y-a.y)+b.x\*(c.x-a.x))\*(c.y-b.y);
8. dy=(c.x-b.x)\*(b.y\*(c.y-a.y)+b.x\*(c.x-a.x))-(c.x-a.x)\*(a.y\*(c.y-b.y)+a.x\*(c.x-b.x));
9. return(point(dx/d,dy/d));
10. }
11. point circumcenter(point a,point b,point c) {// 外心，直角三角形须特判
12. double A=dist(b,c),B=dist(a,c),C=dist(a,b);
13. double P=(SQR(A)+SQR(B)+SQR(C))/2.0;
14. double Q=1.0/(1/(P-SQR(A))+1/(P-SQR(B))+1/(P-SQR(C)));
15. double R=sqrt(P-Q)/2; //R 为外接圆半径，需要时可用，否则可删去
16. double d1=Q/(P-SQR(A)),d2=Q/(P-SQR(B)),d3=Q/(P-SQR(C));
17. return((1-d1)/2.0\*a+(1-d2)/2.0\*b+(1-d3)/2.0\*c);
18. }
19. point incenter(point a,point b,point c) {
20. double A=dist(b,c),B=dist(a,c),C=dist(a,b);
21. double r=2\*area(a,b,c)/(A+B+C); //r 为内切圆半径，需要时可用，否则可删去
22. return(point((A\*a.x+B\*b.x+C\*c.x)/(A+B+C),(A\*a.y+B\*b.y+C\*c.y)/(A+B+C)));
23. }

### 三角剖分

1. /\*
2. Delaunay Triangulation 随机增量算法 :
3. 节点数至少为点数的 6 倍, 空间消耗较大注意计算内存使用
4. 建图的过程在 build 中, 注意初始化内存池和初始三角形的坐标范围 (Triangulation::LOTS)
5. Triangulation::find 返回包含某点的三角形
6. Triangulation::add\_point 将某点加入三角剖分
7. 某个 Triangle 在三角剖分中当且仅当它的 has\_children 为 0
8. 如果要找到三角形 u 的邻域, 则枚举它的所有 u.edge[i].tri, 该条边的两个点为 u.p[(i+1)%3], u.p[(i+2)%3]
9. \*/
10. const int N = 100000 + 5, MAX\_TRIS = N \* 6;
11. const double EPSILON = 1e-6, PI = acos(-1.0);
12. struct Point {
13. double x,y; Point():x(0),y(0){} Point(double x, double y):x(x),y(y){}
14. bool operator ==(Point const& that)const {return x==that.x&&y==that.y;}
15. };
16. inline double sqr(double x) { return x\*x; }
17. double dist\_sqr(Point const& a, Point const& b){return sqr(a.x-b.x)+sqr(a.y-b.y);}
18. bool in\_circumcircle(Point const& p1, Point const& p2, Point const& p3, Point const& p4) {
19. double u11 = p1.x - p4.x, u21 = p2.x - p4.x, u31 = p3.x - p4.x;
20. double u12 = p1.y - p4.y, u22 = p2.y - p4.y, u32 = p3.y - p4.y;
21. double u13 = sqr(p1.x) - sqr(p4.x) + sqr(p1.y) - sqr(p4.y);
22. double u23 = sqr(p2.x) - sqr(p4.x) + sqr(p2.y) - sqr(p4.y);
23. double u33 = sqr(p3.x) - sqr(p4.x) + sqr(p3.y) - sqr(p4.y);
24. double det = -u13\*u22\*u31 + u12\*u23\*u31 + u13\*u21\*u32 - u11\*u23\*u32 - u12\*u21\*u33 + u11\*u22\*u33;
25. return det > EPSILON;
26. }
27. double side(Point const& a, Point const& b, Point const& p) { return (b.x-a.x)\*(p.y-a.y) - (b.y-a.y)\*(p.x-a.x);}
28. typedef int SideRef; struct Triangle; typedef Triangle\* TriangleRef;
29. struct Edge {
30. TriangleRef tri; SideRef side; Edge() : tri(0), side(0) {}
31. Edge(TriangleRef tri, SideRef side) : tri(tri), side(side) {}
32. };
33. struct Triangle {
34. Point p[3]; Edge edge[3]; TriangleRef children[3]; Triangle() {}
35. Triangle(Point const& p0, Point const& p1, Point const& p2) {
36. p[0]=p0;p[1]=p1;p[2]=p2;children[0]=children[1]=children[2]=0;
37. }
38. bool has\_children() const { return children[0] != 0; }
39. int num\_children() const {
40. return children[0] == 0 ? 0
41. : children[1] == 0 ? 1
42. : children[2] == 0 ? 2 : 3;
43. }
44. bool contains(Point const& q) const {
45. double a=side(p[0],p[1],q), b=side(p[1],p[2],q), c=side(p[2],p[0],q);
46. return a >= -EPSILON && b >= -EPSILON && c >= -EPSILON;
47. }
48. } triange\_pool[MAX\_TRIS], \*tot\_triangles;
49. void set\_edge(Edge a, Edge b) {
50. if (a.tri) a.tri->edge[a.side] = b;
51. if (b.tri) b.tri->edge[b.side] = a;
52. }
53. class Triangulation {
54. public:
55. Triangulation() {
56. const double LOTS = 1e6;
57. the\_root = new(tot\_triangles++) Triangle(Point(-LOTS,-LOTS),Point(+LOTS,-LOTS),Point(0,+LOTS));
58. }
59. TriangleRef find(Point p) const { return find(the\_root,p); }
60. void add\_point(Point const& p) { add\_point(find(the\_root,p),p); }
61. private:
62. TriangleRef the\_root;
63. static TriangleRef find(TriangleRef root, Point const& p) {
64. for( ; ; ) {
65. if (!root->has\_children()) return root;
66. else for (int i = 0; i < 3 && root->children[i] ; ++i)
67. if (root->children[i]->contains(p))
68. {root = root->children[i]; break;}
69. }
70. }
71. void add\_point(TriangleRef root, Point const& p) {
72. TriangleRef tab,tbc,tca;
73. tab = new(tot\_triangles++) Triangle(root->p[0], root->p[1], p);
74. tbc = new(tot\_triangles++) Triangle(root->p[1], root->p[2], p);
75. tca = new(tot\_triangles++) Triangle(root->p[2], root->p[0], p);
76. set\_edge(Edge(tab,0),Edge(tbc,1));set\_edge(Edge(tbc,0),Edge(tca,1));
77. set\_edge(Edge(tca,0),Edge(tab,1));set\_edge(Edge(tab,2),root->edge[2]);
78. set\_edge(Edge(tbc,2),root->edge[0]);set\_edge(Edge(tca,2),root->edge[1]);
79. root->children[0]=tab;root->children[1]=tbc;root->children[2]=tca;
80. flip(tab,2); flip(tbc,2); flip(tca,2);
81. }
82. void flip(TriangleRef tri, SideRef pi) {
83. TriangleRef trj = tri->edge[pi].tri; int pj = tri->edge[pi].side;
84. if(!trj||!in\_circumcircle(tri->p[0],tri->p[1],tri->p[2],trj->p[pj])) return;
85. TriangleRef trk = new(tot\_triangles++) Triangle(tri->p[(pi+1)%3], trj->p[pj], tri->p[pi]);
86. TriangleRef trl = new(tot\_triangles++) Triangle(trj->p[(pj+1)%3], tri->p[pi], trj->p[pj]);
87. set\_edge(Edge(trk,0), Edge(trl,0));
88. set\_edge(Edge(trk,1), tri->edge[(pi+2)%3]); set\_edge(Edge(trk,2), trj->edge[(pj+1)%3]);
89. set\_edge(Edge(trl,1), trj->edge[(pj+2)%3]); set\_edge(Edge(trl,2), tri->edge[(pi+1)%3]);
90. tri->children[0]=trk;tri->children[1]=trl;tri->children[2]=0;
91. trj->children[0]=trk;trj->children[1]=trl;trj->children[2]=0;
92. flip(trk,1); flip(trk,2); flip(trl,1); flip(trl,2);
93. }
94. };
95. int n; Point ps[N];
96. void build(){
97. tot\_triangles = triange\_pool; cin >> n;
98. for(int i = 0; i < n; ++ i) scanf("%lf%lf",&ps[i].x,&ps[i].y);
99. std::random\_shuffle(ps, ps + n); Triangulation tri;
100. for(int i = 0; i < n; ++ i) tri.add\_point(ps[i]);
101. Triangle xx = \*tri.find(ps[3]);
102. cout << xx.p[0].x << " " << xx.p[0].y << endl;
103. cout << xx.p[1].x << " " << xx.p[1].y << endl;
104. cout << xx.p[2].x << " " << xx.p[2].y << endl;
105. }

### 凸包的相关操作

1. /\*
2. 给定凸包, $\log n$ 内完成各种询问, 具体操作有 :
3. 1. 判定一个点是否在凸包内
4. 2. 询问凸包外的点到凸包的两个切点
5. 3. 询问一个向量关于凸包的切点
6. 4. 询问一条直线和凸包的交点
7. INF 为坐标范围, 需要定义点类大于号
8. 改成实数只需修改 sign 函数，以及把 long long 改为 double 即可
9. 构造函数时传入凸包要求无重点, 面积非空, 以及 pair(x,y) 的最小点放在第一个
10. \*/
11. const long double PI = acos(-1.0);
12. const double eps = 1e-8;
13. inline int dcmp(double x) { return (x>eps) - (x<-eps); }
14. int sign(double x)
15. {
16. return (x>eps) - (x<-eps);
17. }
18. #define Vector Point
19. struct Point {
20. double x, y; //int belong;//属于哪一个圆
21. Point(double x = 0, double y = 0) : x(x), y(y) {}
22. Vector operator - (const Point& rhs) const {
23. return Vector(x - rhs.x, y - rhs.y); }
24. bool operator < (const Point& rhs) const {
25. return dcmp(x - rhs.x) < 0
26. || (dcmp(x-rhs.x)==0 && dcmp(y-rhs.y) < 0); }
27. bool operator == (const Point& rhs) const {
28. return dcmp(x - rhs.x) == 0 && dcmp(y - rhs.y) == 0; }
29. bool operator > (const Point& rhs) const {
30. return !(\*this < rhs || \*this == rhs); }
31. double det(const Vector& rhs) {
32. return x \* rhs.y - y \* rhs.x; }
33. Point in() { scanf("%lf%lf", &x, &y); return \*this; }
34. void out() const { printf("(%lf, %lf) ", x, y); }
35. };
36. struct Convex
37. {
38. int n;
39. vector<Point> a, upper, lower;
40. Convex(){}
41. Convex(vector<Point> \_a) : a(\_a) {
42. n = a.size();
43. int ptr = 0;
44. for(int i = 1; i < n; ++ i) if (a[ptr] < a[i]) ptr = i;
45. for(int i = 0; i <= ptr; ++ i) lower.push\_back(a[i]);
46. for(int i = ptr; i < n; ++ i) upper.push\_back(a[i]);
47. upper.push\_back(a[0]);
48. }
49. int sign(long long x) { return x < 0 ? -1 : x > 0; }
50. pair<long long, int> get\_tangent(vector<Point> &convex, Point vec) {
51. int l = 0, r = (int)convex.size() - 2;
52. for( ; l + 1 < r; ) {
53. int mid = (l + r) / 2;
54. if (sign((convex[mid + 1] - convex[mid]).det(vec)) > 0) r = mid;
55. else l = mid;
56. }
57. return max(make\_pair(vec.det(convex[r]), r), make\_pair(vec.det(convex[0]), 0));
58. }
59. void update\_tangent(const Point &p, int id, int &i0, int &i1) {
60. if ((a[i0] - p).det(a[id] - p) > 0) i0 = id;
61. if ((a[i1] - p).det(a[id] - p) < 0) i1 = id;
62. }
63. void binary\_search(int l, int r, Point p, int &i0, int &i1) {
64. if (l == r) return;
65. update\_tangent(p, l % n, i0, i1);
66. int sl = sign((a[l % n] - p).det(a[(l + 1) % n] - p));
67. for( ; l + 1 < r; ) {
68. int mid = (l + r) / 2;
69. int smid = sign((a[mid % n] - p).det(a[(mid + 1) % n] - p));
70. if (smid == sl) l = mid;
71. else r = mid;
72. }
73. update\_tangent(p, r % n, i0, i1);
74. }
75. int binary\_search(Point u, Point v, int l, int r) {
76. int sl = sign((v - u).det(a[l % n] - u));
77. for( ; l + 1 < r; ) {
78. int mid = (l + r) / 2;
79. int smid = sign((v - u).det(a[mid % n] - u));
80. if (smid == sl) l = mid;
81. else r = mid;
82. }
83. return l % n;
84. }
85. // 判定点是否在凸包内，在边界返回 true
86. bool contain(Point p) {
87. if (p.x < lower[0].x || p.x > lower.back().x) return false;
88. int id = lower\_bound(lower.begin(), lower.end(), Point(p.x, -INF)) - lower.begin();
89. if (lower[id].x == p.x) {
90. if (lower[id].y > p.y) return false;
91. } else if ((lower[id - 1] - p).det(lower[id] - p) < 0) return false;
92. id = lower\_bound(upper.begin(), upper.end(), Point(p.x, INF), greater<Point>()) - upper.begin();
93. if (upper[id].x == p.x) {
94. if (upper[id].y < p.y) return false;
95. } else if ((upper[id - 1] - p).det(upper[id] - p) < 0) return false;
96. return true;
97. }
98. // 求点 p 关于凸包的两个切点，如果在凸包外则有序返回编号，共线的多个切点返回任意一个，否则返回 false
99. bool get\_tangent(Point p, int &i0, int &i1) {
100. if (contain(p)) return false;
101. i0 = i1 = 0;
102. int id = lower\_bound(lower.begin(), lower.end(), p) - lower.begin();
103. binary\_search(0, id, p, i0, i1);
104. binary\_search(id, (int)lower.size(), p, i0, i1);
105. id = lower\_bound(upper.begin(), upper.end(), p, greater<Point>()) - upper.begin();
106. binary\_search((int)lower.size() - 1, (int)lower.size() - 1 + id, p, i0, i1);
107. binary\_search((int)lower.size() - 1 + id, (int)lower.size() - 1 + (int)upper.size(), p, i0, i1);
108. return true;
109. }
110. // 求凸包上和向量 vec 叉积最大的点，返回编号(0下标开始)，共线的多个切点返回任意一个
111. int get\_tangent(Point vec) { //凸包向量 ^ vec
112. pair<long long, int> ret = get\_tangent(upper, vec);
113. ret.second = (ret.second + (int)lower.size() - 1) % n;
114. ret = max(ret, get\_tangent(lower, vec));
115. return ret.second;
116. }
117. // 求凸包和直线 u,v 的交点, 如果无严格相交返回 false. 如果有则是和 (i,next(i)) 的交点, 两个点无序, 交在点上不确定返回前后两条线段其中之一
118. bool get\_intersection(Point u, Point v, int &i0, int &i1) {
119. int p0 = get\_tangent(u - v), p1 = get\_tangent(v - u);
120. if (sign((v - u).det(a[p0] - u)) \* sign((v - u).det(a[p1] - u)) < 0) {
121. if (p0 > p1) swap(p0, p1);
122. i0 = binary\_search(u, v, p0, p1);
123. i1 = binary\_search(u, v, p1, p0 + n);
124. return true;
125. } else {
126. return false;
127. }
128. }
129. };