[字符串处理 3](#_Toc478335874)

[1、kmp算法 3](#_Toc478335875)

[2、扩展kmp 4](#_Toc478335876)

[3、字符串最小表示法 6](#_Toc478335877)

[4、Manacher 7](#_Toc478335878)

[5、字典树 9](#_Toc478335879)

[6、最小包含子串 11](#_Toc478335880)

[7、Aho-Corasick automaton 14](#_Toc478335881)

[8、字符串Hash，找区间不同子串个数 17](#_Toc478335882)

[9、Suffix Array 20](#_Toc478335883)

[杂项 22](#_Toc478335884)

[数据结构 22](#_Toc478335885)

[1、并查集 22](#_Toc478335886)

[2、线段树 26](#_Toc478335887)

[3、树状数组 27](#_Toc478335888)

[4、RMQ 30](#_Toc478335889)

[5、单调队列 & 单调栈 30](#_Toc478335890)

[6、分块 31](#_Toc478335891)

[杂项 32](#_Toc478335892)

[动态规划 33](#_Toc478335893)

[1、整数划分 33](#_Toc478335894)

[2、区间DP 34](#_Toc478335895)

[3、数位DP 34](#_Toc478335896)

[4、状压DP 36](#_Toc478335897)

[杂项 37](#_Toc478335898)

[图论 38](#_Toc478335899)

[1、最短路径 39](#_Toc478335900)

[2、最小生成树（MST） 42](#_Toc478335901)

[3、LCA、树的重心、树的直径 43](#_Toc478335902)

[4、图的割点、割边、tarjan 47](#_Toc478335903)

[5、二分图匹配 49](#_Toc478335904)

[6、欧拉回路 && 哈密顿回路 52](#_Toc478335905)

[杂项 53](#_Toc478335906)

[计算几何 54](#_Toc478335907)

[1、基本公式 54](#_Toc478335908)

[2、正N边形公式 58](#_Toc478335909)

[3、平面最近对 59](#_Toc478335910)

[4、欧拉公式，分割平面 61](#_Toc478335911)

[数学 61](#_Toc478335912)

[1、组合数Cnm 防溢出公式 61](#_Toc478335913)

[2、各种素数筛法 65](#_Toc478335914)

[3、快速幂、矩阵 67](#_Toc478335915)

[4、数字特征、约数个数 69](#_Toc478335916)

[5、扩展欧几里德算法和求逆元 71](#_Toc478335917)

[6、各种数列（卡特兰数、 递推式组合数） 74](#_Toc478335918)

[7、米勒测试和大数分解 75](#_Toc478335919)

[8、欧拉函数eular 77](#_Toc478335920)

[9、AntiPrime 78](#_Toc478335921)

[10、万能积分公式---simpson 80](#_Toc478335922)

[11、高斯消元 81](#_Toc478335923)

[杂项 82](#_Toc478335924)

[其他 86](#_Toc478335925)

[1、STL 86](#_Toc478335926)

[2、常量定义 & 手动开栈 & C++取消同步 & int范围 88](#_Toc478335927)

[3、三分答案 89](#_Toc478335928)

[4、最长下降子序列 89](#_Toc478335929)

[5、高精度、输入挂、java大数 90](#_Toc478335930)

[6、基本思考方式 93](#_Toc478335931)

[杂项 94](#_Toc478335932)

# 字符串处理

## 1、kmp算法

主串：匹配串。 子串：模式串。

子串先自己匹配，得到next[]，然后再和主串进行匹配，匹配时主串的i值不回溯，所以使得整个算法的复杂度压缩为O(lenstr+lensub),关于next[]数组的意义:next[i]的意思是，匹配到当前第i个字符，如果那个字符和主串当前的匹配字符不搭配，那么就跳转到sub[]中的第next[i]个字符匹配。为什么能这样做呢？那是因为子串中，前缀(1--next[i]-1)和后缀(begin--i-1)是完全相等的，begin的值各不相同，例如abcabc中，next[len]=3(不包括最后那个c的)，那么就是[1--2]和[4,5]是完全相等的，next[len+1]=4,也就是[1--3]和[3--6]是完全相等的。那么这样的话，next[i]的意思就是在长度为i-1中的子串中，最长的前缀-后缀的长度为next[i]-1。因为next[i]是告诉我们应该跳去哪里匹配，是"越界"的。

**void** get\_next(**char** sub[], **int** tonext[], **int** lensub) {

**int** i = 1, j = 0;

tonext[1] = 0; //记得初始值不能忘记

//tonext[lensub + 1]这个也有值了 后面的++i和++j先加后赋值

**while** (i <= lensub) {

//sub[i]的含义，后缀的单个字符 //sub[j]的含义，前缀的单个字符

**if** (j == 0 || sub[i] == sub[j]) { //考虑的是上一个的

tonext[++i] = ++j; //用是上一个的比较，值的当前的值

} **else** j = tonext[j];

}

}

返回主串str[]中子串sub[]的出现次数，(允许重叠的部分),这里的重叠部分就是说：例如"aaa",则"aa"出现的次数为2次。 pos为在主串的第pos个位置开始匹配，默认为1

**int** kmp(**char** str[], **char** sub[], **int** pos) {

**int** lenstr = strlen(str + 1);

**int** lensub = strlen(sub + 1);

**int** tonext[maxn] = {0}; //maxn为最大长度

get\_next(sub, tonext, lensub); //得到next[]数组

**int** i = pos; //从哪里出发，i变量是主串的。

**int** j = 1; // j变量是子串的。

**int** ans = 0;

**while** (i <= lenstr) {

**if** (j == 0 || str[i] == sub[j]) {

i++;

j++;

} **else** j = tonext[j];

**if** (j == lensub + 1) { //有一个了

ans++;

j = tonext[j]; //回溯匹配

//i值不用回溯的

}

}

**return** ans;

}

KMP求循环节： **1 <= cir < lenstr**

当next[lenstr + 1] = 1时，**求出来的并不是循环节！**此时cir = lenstr

cir=lenstr-(next[lenstr+1]-1); // 最小循环节的长度,什么时候都成立，不够可以补ab\*\*\*ab(\*\*\*)

anstime = len / cir; (只有len % cir == 0时成立，只有能整除，才能写成cir ^ k )

如果是这样的: [abc**abcab**]**cab，**这样求出的cir还是3的，然后出现次数是floor(lenstr / cir)

求cir = 3是否这个串的最小循环节，可以删除一个字符。例如：a@bcabcabc

要判断是否每三个循环，只需str[1] == str[4] && str[2] == srt[5] && str[3] == str[6]

如果某一个不满足，则重新开始。然后找到了lenstr个满足，那么这个cir就是满足的，然而这里还是得不到答案，那么我们把这个串连续写两次。a@**bcabcabc a**@bcabcabc

就能把中间那段扫出来，因为循环节旋转后，还是循环的。

## 2、扩展kmp

扩展kmp (下标从1开始)

求解主串str[i--n]中，和给定的子串sub[]的最长公共前缀，要求在线性时间内找出所有的 extend[i]表示:主串的第i位开始到n(主串长度)，和子串sub[]的最长公共前缀。(一定要从头开始，前缀嘛)

解法：设a为已经求解出来的extend[i]中，使得匹配距离最大(i+extend[i]-1)的i(是一个下标),这样的匹配数是最优的，我们去判断有没更好的匹配时，如果比这个max值(记作p)还小，那么就不用匹配啦!!!那么根据extend[i]的定义：和sub[]有相同前缀嘛！！就有str[a...p]=sub[1...p-a+1](区间长度要相等)

对于现在每一个待求的值k(易得k>a,但是不一定大于p)，

有str[k...p]=sub[k-a+1...p-a+1](区间长度要相等,利用区间长度相等，能反解出k-a+1)但是我求的是前缀，你这样给我和后面的相等没用，这样设函数next[i]表示sub[]中，i—m(子串长度)和sub[]整个串的最大公共前缀(和上面那个意义一样).则有设L=next[k-a+1];

有sub[k-a+1....??]=sub[1...L](这个区间长度是L);这样可以分类讨论

①、如果(k-1)+L<p，就是这个区间从k开始覆盖，若不能覆盖到p，说明后面的不用比较了，不等。

这个时候extend[k]=L;(注意这里的<不能相等，等于p表明区间大小一样，但是后面的能不能匹配，我们还不知道，next[k-a+1]告诉我们的只是它自身的匹配，现在我的是str和sub的匹配，概念不同)

②、else 则要比较str[p+1]和sub[L+1]位，一直比较下去，更新a值即可

**void** get\_pre\_next(**char** sub[], **int** lensub, **int** next[]) { //数组的下标从1开始

next[1] = lensub; //固定的大小

**int** a = 1;

**while** ((a + 1) <= lensub && sub[a] == sub[a + 1]) {

a++;

}

next[2] = a - 1; //算多了一个，一开始a本来自定义有一个

a = 2; //开始时，能得到最大值p的就是2啦。

**for** (**int** k = 3; k <= lensub; k++) { //现在求解的是k

**int** p = a + next[a] - 1; //p是能达到的最远距离那个字母的下标，

//易得sub[a...p] = sub[1...p-a+1]，又因为sub[k…p]是sub[a…p]的子串

//所以sub[k..p] = sub[k-a+1..p-a+1]。加粗的部分可以用区间长度相等得到

**int** L = next[k - a + 1]; //next[]的是加粗的部分，这样的话，sub[k…??] = sub[1..L]

//现在求解的是K,不是k+1

**if** ((k - 1) + L < p) { //不能取等，这个覆盖是根据sub[k…??] = sub[1..L]来覆盖的。

next[k] = L; //如果这都覆盖不到去p，那么最多只能是L

//因为我们是用了next[]来得到L的。如果还能继续匹配，那么就违背了next[]的定义了

//如果sub[??+1]==sub[L+1]的话，因为sub[k..p]=sub[k-a+1,p-a+1]，而sub[??+1]在sub[k..p]中，

//那么就会有对应的字母在sub[k-a+1,p-a+1]中，那么你next[k-a+1]得到的会是L+1而不是L

} **else** { //如果是相等的话，我还能暴力看看sub[p+1]和sub[L+1]是不是相等的。

//现在的：目的是比较sub[p+1]和sub[L+1]

**int** j = max(0, p - k + 1); //区间长度

//用了个区间长度变量，方便用来和后面的比较

//区间起点是k,那么k+j就是p+1 (这个代入去max那里得到) ，或者是sub[k]本身，p的值//不一定比k大的，例如”abcabcabc”中的第四个a的时候。p-k+1=-1，所以要用max来修正。

//而1+j，由if的条件，L>p-k+1，那么这是和sub[p+1]匹配的下一个sub[p-k+2]，或者是sub[1]

**while** ((k + j) <= lensub && sub[k + j] == sub[1 + j]) {

//这一个1+是相对于起始位置1

j++;

}

next[k] = j;

a = k; //改变最优值，这个值必须变大k变大了，p也变大了

}

}

**return** ;

}

//求next[]和上面的差不多，但是上面的解释只是用来求extend[]的。

**void** extend\_kmp(**char** str[], **char** sub[], **int** extend[]) {

**int** lenstr = strlen(str + 1);

**int** lensub = strlen(sub + 1);

**int** next[maxn] = {0};

get\_pre\_next(sub, lensub, next);

**int** a = 1;

**while** (a <= lenstr && a <= lensub && str[a] == sub[a]) {

a++;

}

extend[1] = a - 1; //同样开始的时候认为它是1 所以多了一个

a = 1; //假设1是现在达到的最大的p

**for** (**int** k = 2; k <= lenstr; k++) { //现在求解的是k

**int** p = a + extend[a] - 1; //能匹配到的最远值的[下标]

//注意这里是用extend[]的，其实求next[]和extend[]一个意思

//extend[]是str[]和sub[]的 next[]是sub[]和sub[]的

**int** L = next[k - a + 1];

**if** ((k - 1) + L < p) { //不能取等

extend[k] = L;

} **else** {

**int** j = MAX(p - k + 1, 0); //区间长度

**while**((k + j) <= lenstr && (1 + j) <= lensub && str[k + j] == sub[1 + j]) {

j++;

}

extend[k] = j;

a = k;

}

}

**return** ;

}

char str[100]="aaaabcdaab"; char sub[100]="aaabcd"; 说明不能取等的数据

## 3、字符串最小表示法

给定一个字符串，是首尾相接的，要求找到一个位置pos，使得遍历这个字符串，其字典序是最小的。例如”1100” anspos=3

\*\*数据量巨大(N <= 1000000),显然只能用O(n)的算法。

1、朴素的比较算法，设一个指针为ans，一个为比较指针cmp，找到

str[(ans + k) % lenstr] != str[(cmp + k) % lenstr],如果前者较大，则ans = cmp，表明在cmp开始的字符串字典序更小，一路下去。复杂度为O(n²)。这里的缺陷就是，每次比较完后，ans指针的移动量太小了，才移动到去cmp那里，最坏情况下，只是一格一格移动的而已，而每次移动都要比较整个串的长度。例如:bbbbbbba

2、较优秀的匹配算法，主要解决了ans指针的移动距离问题，其实可以看到，如果str[(ans+k)%lenstr]>str[(cmp+k)%lenstr],那么，在str[ans—ans+k]中，是不会存在答案的，也就是不会存在最小表示法在那些位置中，证明如下：假如存在，设那个位置为x(ans<=x<=ans+k)，那么很明显，因为有str[(ans+k)%lenstr]>str[(cmp+k)%lenstr],所以我能找到一个串，位置为Y，在它匹配的时候，又匹配ans+k那个位置，使得它又失配了。

ans指针：那个小了，就跳去哪一个，比如66664，第一步会跳去4

cmp指针：那个大了，就跳去大了的下一个。比如222262，cmp跳去最后一个2

ans x ans+k jumpHere

6 2 3 6 2 3 6 2 3 4

6 2 3 6 2 3 6 2 3 4

cmp Y cmp+k

所以，当ans失配的时候，直接跳到ans+k+1的地方，继续开始比较，同理当cmp比较大的时候，直接跳就可以了，然后，取min(ans,cmp),就是答案，比较指针也是有保存答案值的。(但是我现在也没找到这样的例子，看到网上是这样说，我做题的时候直接放回ans也ac了)

\*\*另外一个要注意的地方是，可能，ans移动的时候，直接移动到cmp的那个地方了，这个时候，cmp就要+1，因为要找下一个地方比较嘛。

**int** get\_min\_pos(**char** str[], **int** lenstr) {

**int** ans = 1, cmp = 2;

**while** (ans <= lenstr && cmp <= lenstr) {

**int** k, t1, t2;

**for** (k = 0; k <= lenstr - 1; k++) { //匹配整个串的长度就够

t1 = ans + k;

t2 = cmp + k;

**if** (t1 > lenstr) t1 %= lenstr; //下标从1开始要这样%

**if** (t2 > lenstr) t2 %= lenstr;

**if** (str[t1] != str[t2]) **break**;

}

//如果匹配了整个串的长度了，证明找不到更小的了

**if** (k == lenstr) **break**;

**if** (str[t1] > str[t2]) { //最大表示直接改符号就可以

ans += k + 1;

} **else** {

cmp += k + 1;

}

**if** (ans == cmp) cmp++; //找下一个比较的地方

}

**return** min(ans, cmp); //这个好像不返回min也可以AC?

}

## 4、Manacher

给定一个字符串，要求出里面最长的回文子串。\*\*\*abcba\*\*\*

思路：记p[i]为以i为原点，半径为p[i]的最长回文子串。记id为当前已算出的p[i]中，能去到的最远距离的下标。就是max(p[i]+i)。那么，如果当我们求解i的时候，如果被以前求出的id包围了，那么，根据回文串的对称性，点i关于点id对称的点是2\*id-i，（这个能根据i-id+1 == id-x+1）区间相等，求出。因为p[2\*id-i]是已经求出来的了，①、如果p[2\*id-i]超越了p[id] 的范围，则只能去到p[id]+id-i，如果还能继续匹配，就跟p[id]矛盾。（左边匹配比较大，不代表右边也能匹配那么多，就是p[2\*id-i]和p[i]的大小没必然联系），如果右边也还能匹配，那么p[id]应该变大，矛盾。

②、如果没超越，那么就是p[2\*id-i]的值了。

如果端点重合了，则可能继续判断，右边可能匹配更多。所以，就需要暴力判断了。

\*\*解决”aa”这样的偶数回文串中心不知道是那个，我们用’#’隔开每个字符,同时注意str[0]和str[2\*lenstr+2]不能相同。一共插入了lenstr+1个’#’,使得长度变成2\*lenstr+1

**数组记得要开2\*maxn**

**字母所在位置：都是偶数的。 ’#’所在的位置：都是奇数的。一般要分类讨论。**

**int** manacher(**char** str[], **int** lenstr) {

str[0] = '\*'; //表示一个不可能的值

//目标要插入lenstr+1个'#'，所以长度变成2\*lenstr+1

**for** (**int** i = lenstr; i >= 0; i--) { //str[lenstr+1]是'\0'

//i=lenstr时，i+i+2那个值要赋为'\0';

//总长度只能是lenstr+lenstr+2,所以i从lenstr开始枚举

str[i + i + 2] = str[i + 1];

str[i + i + 1] = '#';

}

**int** id = 0, maxlen = 0; //现在开始在str[2]了

**for** (**int** i = 2; i <= 2 \* lenstr + 1; i++) { //2\*lenstr+1是'#'没用

**if** (p[id] + id > i) { //没取等号的，只能去到p[id]+id-1

//p[id]+id是越界的，减去i即为区间长度

//p[id]+id-i,这个是所有可能中的最大值了

p[i] = min(p[id] + id - i, p[2 \* id - i]);

} **else** p[i] = 1; //记得修改的是p[i]

**while** (str[i + p[i]] == str[i - p[i]]) ++p[i];

**if** (p[id] + id < p[i] + i) id = i;

maxlen = max(maxlen, p[i]);

//为什么maxlen是p[i]而不是2\*p[i]-1 呢??

//因为\*#a#b#a#在b中，开始时候p[i]是1(有效值)，扩展p[i]=2

//但是这个增加是无效值，又p[i]=3(有效值),.....每次扩展一个有效值，必能扩展一个无效值，//而且一定是无效值结尾，所以ans=maxlen-1减去1是为了减去最后一个无效值。

//p[i]是包括#号字符串的回文串的半径（包括自己），扩展一个有效值后，又扩展一个无效

//值(相当于数量增加的是左边的有效值)，所以p[i]就是回文串长度

}

**return** maxlen - 1;

}

小变形题目 ---- 吉哥系列故事――完美队形II

题目：在一个主串 maxn=100000 中，求回文子串，但是，那个回文子串是要递增的。就是

1234321这样才行，h[1]<=h[2]<=h[3]<=h[mid]

思路：其实可以直接更改p[i]的意义即可。记p[i]为以i为原点，p[i]为半径的“回文串”长度，这个回文串是满足题目条件的回文串。你可能会认为这样的p[i]值可能会很小，回文串长度是多，但是满足这个条件的回文串长度很少啊。答案是：少也没问题的，你的意思是p[id]+id覆盖不了你的i是吧？那么就p[i]=1啊，暴力判断啊。我们用p[i]就是为了加快而已，并不影响答案的最优性。

while (str[i+p[i]]==str[i-p[i]] && num[i-p[i]]<=num[i-p[i]+2]) ++p[i];

每增加一个数，和它的隔壁的隔壁比较，因为我们用inf隔开了嘛！！

0 inf 1 inf 2 inf 3 inf 2 inf 1 inf -1

## 5、字典树

\*\*如果字符串从1开始，判断和字符串”END”是否相等，是： **strcmp(str+1,”END”)==0**

**\*\*用字典树时记得看清楚那些字母是数字还是字母，因为压缩下标时减去的数是不同的**

字典树解决的是多模式匹配问题，给定一个串，要求在单词本里问有没出现这个串，或者有没出现以这个串为前缀的串等等。明显的，如果我要找”abc”，那么以b开头的串我肯定是不必找的了，所以每次只需匹配每个字母，查找的复杂度为O（lenstr）,插入的复杂度也是O（lenstr）。每个节点都包含26枚指针（小写字母个数，如果包含大写字母，则52即可，int id的时候判断一下大小写，小写的话减去‘a’+偏移位置26）。空间换时间~~

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| “a” 0 2 | “b” 0 2 | “c” 1 1 | “d” 1 1 | ch, flag, count |

指向b NULL 指向c 指向d

//注意这里字符串的开始位置是从1开始

**const** **int** maxn = 1e5;

**const** **int** N = 26; //26个小写字母

**struct** node {

**int** flag; //标记以这个字母结尾为一个单词

**int** count; //标记以这个字母结尾为一个前缀

**struct** node \*pNext[N]; //26枚字符指针

} tree[maxn \* N]; //大小通常设为 单词个数\*单词长度

**int** t;//表明现在用到了那个节点

**struct** node \*create() {

//需要新开一个字符节点，就是有abc这样，插入abd，则d需要新开节点

**struct** node \*p = &tree[t++];

p->flag = 0; //初始值为0，不是整个单词

p->count = 1; //前缀是必须的，本身就是一个了

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

p->pNext[i] = NULL; //初始化指针

}

**return** p;

}

**void** insert(**struct** node \*\*T, **char** str[]) {

**struct** node \*p = \*T;

**if** (!p) { //空树

p = \*T = create();

}

**int** lenstr = strlen(str + 1);

**for** (**int** i = 1; i <= lenstr; i++) { //把单词拆开放进树

**int** id = str[i] - 'a'; //压缩下标

**if** (p->pNext[id]) { //存在过，则前缀++

p->pNext[id]->count++; //p->pNext[id]表明是id这个字母

} **else** {

p->pNext[id] = create();

}

p = p->pNext[id];

}

p->flag = 1; //表明这字母为结尾是一个单词，上一次已经是p=p->pNext[id]了

//就是现在已经去到了单词的最后一个字母的那个节点了！！

**return** ;

}

**int** find(**struct** node \*T, **char** str[]) {

**struct** node \*p = T;

**if** (!p) { //空树

**return** 0;

}

**int** lenstr = strlen(str + 1);

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

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

**if** (!p->pNext[id]) {

**return** 0; //单词中断，找不到

}

p = p->pNext[id];

}

**return** p->flag; //看看是不是一个单词的结尾。也可以返回是否前缀

}

字典树解决异或的题目的时候。

1、一般可能会用到前缀异或和，然后转化为数组中两个数异或最大值。

2、插入数字的时候，判断第i位是否为1，记得判断是否大于0，((1 << i) & val) > 0

3、删除某一个数字，可以把这条路的cnt都减去1，然后查找的时候判断下cnt即可。

例题：HDU 5687 (百度之星)

删除前缀为str[]的**所有单词**

这个cut可以直接用find(T,str)来找到以str为前缀出现了多少个单词。就是：return p->count;

例如插入了hello 和 helloliu。然后要删除hello为前缀的单词。那么：因为每个h节点，数量是为2的，**如果**又插入了haha这样的话，h数量为3。那么，要删除hello为前缀的单词，每个h应该减去以它为前缀出现的次数，也就是减去2了。减去1的话，search h 会是Yes。（这是没有插入haha的前提下，search h 还是 Yes）

**void** did(**struct** node \*\*T, **char** str[], **int** cut) {

**if** (cut == 0) {

**return** ; //不存以这个为前缀的单词。

}

**struct** node \*p = \*T;

**if** (!p) **return** ;

**int** lenstr = strlen(str + 1);

**for** (**int** i = 1; str[i]; i++) { //新姿势，直接用str[i]结束符判断，不用测量长度了！

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

p = p->pNext[id]; //现在跳去这个字母的节点

p->count -= cut;//删除无非就是把以这个字符串为前缀开头的数目减掉。

}

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

p->pNext[i] = NULL; //把它的孩子统统杀死

}

p->flag = 0;

**return** ;

}

## 6、最小包含子串

给定一个主串str[]和一个子串sub[]，要求得在主串str[]中的一个子串，包含了sub[]的所有字符，顺序可以不同，但是数目必须是>=sub[]中的。

思路：直接two pointer模拟，判断是否出现可以直接hash，然后有些注意的地方就是，开头的有些字符串可以省略，然后移动start的时候也有些技巧，就是直接用start移动，因为一头一尾必然是那个sub[]串中的字母(根据贪心策略得到)，所以用start++，就是把第一个有效字母删除，继续找后面的字母代替这个字母。

1、如果找到，那么可能结果更优。因为我可能又省去了前面的一些部分。

2、如果找不到，那么就说明只有那个部分能组合成答案。因为这里字母才足够。

**int** min\_window\_sub(**char** str[], **char** sub[], **char** ans[]) { //51NOD 1127

**int** lenstr = strlen(str + 1);

**int** lensub = strlen(sub + 1);

**int** bookstr[256] = {0}; //字母ASCII最多也就去到128

**int** booksub[256] = {0};

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

booksub[sub[i]]++; //先预处理出所有的sub的hash

}

**int** begin = -inf, end = -inf; //用来赋值给ans[]的，就是答案区间，只是记录答案的而已

**int** found = 0; //表明现在找到了多少个

**int** minlength = lenstr; //最小的长度是整个【主串】

**int** start = 1; //记录从[start,i]中是一个可能性答案

**for** (**int** i = 1; i <= lenstr; i++) { //扫描整个主串

bookstr[str[i]]++;

**if** (bookstr[str[i]] <= booksub[str[i]]) {

//如果这个字符加入后，数目小于这个在子串的次数，那么就是找到了一个字符了

//就是，str="aaaabc" sub="abc"，那么，第一a，说明找到了一个a，第二个a，

//不能说明又找到了一个字符了，只是重复而已。不然就是说又找到了一个字符，wa

found++;

}

**if** (found == lensub) { //找到了足够的字符了

//将开头有些多余的可以去掉 str="aaaaabc" sub="abc"

**while** (start <= i && bookstr[str[start]] > booksub[str[start]]) { //严格大于

bookstr[str[start]]--;

start++;

}

**if** (minlength > i - start + 1) {

minlength = i - start + 1;

begin = start;

end = i;

}

//把开头的这个字符删掉，去找其他可能的情况

//str="a\*\*\*\*\*bc" + “a” sub="abc" 这样的话，\*\*\*可以去掉

bookstr[str[start]]--;

found--;//匹配数减1

start++;

}

}

**if** (begin == -inf) **return** -1; //不存在

**int** t = 1;

**for** (**int** i = begin; i <= end; i++) {

ans[t++] = str[i];

}

**return** end - begin + 1;

}

变形例题：codeforce Educational Codeforces Round 5 **D: Longest k-Good Segment**

题目：给出一个包含n（n<=5e5）的数组，要你找出其中最长的，连续的一段数字，使得其中包含的不同数字的个数不多于k个。

思路：这是一题two pointer的思想，设置两个指针模拟即可，按照最小包含子串这样的来模拟，注意的是判断一个数字有没出现过，用int book[]来保存有没出现过，因为他可能出现很多次，不能用简单的0和1来表示有没出现，因为删除这个数字的时候，如果你book[val]=0就不行了，因为可能这个区间[begin,end]中还包含这个val呢。

#include <cstdio>

#include <cstdlib>

#include <cstring>

const int maxn=5e5+20;

int a[maxn];

int book[1000200];//1e6

void work ()

{

int n,k;

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

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

{

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

}

int head=1,maxlen=0,found=0;

int begin,end;

a[n+1]=1e6+20;//取一个不可能的值，防止边界都成立的情况

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

{

if (!book[a[i]]) //如果这个值没出现过，那么说明是不同数字

{

found++;

}

if (found>k) //如果出现次数大于k了。

//等于k是没用的，因为后面如果有相同的数字，等于k这样的话后面的是统计不进去的。

//例如k=4。1 2 3 4 4 4，第一个4的时候，就已经跳进来了。

{

//i是越界的

if (i-head>maxlen)

{

maxlen=i-head;

begin=head;

end=i-1;

}

//把开头的【重复的数字去掉】，只去掉一个是没用的，这个求的是最长，不是求最短

//求最短的时候，开头相同的优先砍掉了，所以开头的第一个字符就是有效字符。

//而现在求最长，开头的有重复，不全部砍掉的话，那个元素还是存在的。

//所以对于这样的数据 1 1 1 2 3 4 head由1-->4

int num=a[head];

while (head<=i && num==a[head])

{

book[num]--;//减去这些元素出现的次数

head++;

}

if (book[num]) //如果那个元素还是存在呢？就 好像1 1 1 【2 1 3 4】k=3这样

{

//因为现在i是越界的。found==k+1

i--;//从4那个数字开始，再去匹配这个越界的元素

found--;//这个减去的是越界的那个元素

continue; //跳过后面那个东西的而已。

//注意不要再去book[a[i]]++了，这样的话越界的上一个元素统计多一次了

}

else found--;//如果不存在就最好了，直接减去那个元素

}

else //如果不同的元素<=k个的话，那么绝对是可行的解

{

//例子k=2 1 1 1 1 1

if (i!=n+1&&i-head+1>maxlen) //i==n+1这个不是

{

//更新maxlen必须要这个区间长度大于现在的maxlen才行..

//不然的话有wa数据，前一段区间5 1 1 1 1 (k==2)，然后中间什么都行，

//后一段区间 5 1 1 found==k时,进来了，错误地更新了最大值

begin=head;

end=i;

maxlen=i-head+1;

}

}

book[a[i]]++; //这个元素出现次数+1

}

printf ("%d %d\n",begin,end);

return ;

}

int main ()

{

work ();

return 0;

}

## 7、Aho-Corasick automaton

这玩意其实就是trie + kmp，只不过是多串匹配的kmp罢了。next[]把它称为失败指针，当在这个位置匹配失败的时候，就要跳去最大的前后缀那里继续匹配，看看有没和它相同的。以前说的next[]是越界的（指向下一个的），但是这里不能这样，因为下一个是那里?不知道，有26种可能，所以这个不是越界的。所以构造失败指针的时候方法也一样。不断地回溯，例如要求p->next[id]->Fail。那么，先找到p->Fail是哪里，就是爸爸的失败指针，然后再看看哪里有没有字符id，有的话，就是跳去那里了，这样是最优的。就是先找爸爸的兄弟有没这个字符id，爸爸的失败指针表明那里有它的那个字符，再加上是否有我。就继承了下来。

**ac自动机上的等价态：**

等价态即用fail指针连接的点，在行走fail指针时匹配的字符数量并没有发生变化，因此这些点可以看成是相同的匹配状态。通常有两种方法处理等价态：

第一是互为等价态的点各自记录各自的信息。匹配的时候需要遍历所有等价态以判断是否匹配成功。next指针可能为空，需要匹配时进行判断是否需要走fail指针。

第二是所有等价态中的点记录本身以及所有比它浅的点的信息总和（匹配成功的单词总数），匹配时不需要走等价态以判断匹配成功与否。next指针不为空，直接指向本应通过fail指针寻找到的那个状态。

入门题必须A掉：HDU 2222 复杂度O(len \* 单词平均长度)

题目：给定n(n<=10000)个单词，和一篇文章len<=1000000。要求在其中找到有多少个单词是出现过的。一个单词出现多次只算一次。

思路：模板题不解释，AC自动机本来就是解决这类问题的

trick：如果给出两个单词一模一样的，那么应该按照不同串来处理。坑死了~~

存在的点，建立Fail指针。不存在的点，修改p->pNext[id]建立虚拟边。

#include <cstdio>

#include <cstdlib>

#include <cstring>

**const** **int** maxn = 1000000 + 20;

**const** **int** N = 26;

**struct** node {

**int** flag;

**struct** node \*Fail; //失败指针，匹配失败，跳去最大前后缀

**struct** node \*pNext[N];

} tree[maxn];

**int** t; //字典树的节点

**struct** node \*create() { //其实也只是清空数据而已，多case有用

**struct** node \*p = &tree[t++];

p->flag = 0;

p->Fail = NULL;

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

p->pNext[i] = NULL;

}

**return** p;

}

**void** insert(**struct** node \*\*T, **char** str[]) {

**struct** node \*p = \*T;

**if** (p == NULL) {

p = \*T = create();

}

**for** (**int** i = 1; str[i]; i++) {

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

**if** (p->pNext[id] == NULL) {

p->pNext[id] = create();

}

p = p->pNext[id];

}

p->flag++; //相同的单词算两次

**return** ;

}

**void** BuiltFail(**struct** node \*\*T) {

//根节点没有失败指针,所以都是需要特判的

//思路就是去到爸爸的失败指针那里，找东西匹配，这样是最优的

**struct** node \*p = \*T; //用个p去代替修改

**struct** node \*root = \*T;

**if** (p == NULL) **return** ;

//树上bfs,要更改的是p->pNext[i]->Fail

**struct** node \*que[t + 20]; //这里的t是节点总数，字典树那里统计的，要用G++编译

**int** head = 0, tail = 0;

que[tail++] = root;

**while** (head < tail) {

p = que[head]; //p取出第一个元素 ★

**for** (**int** i = 0; i < N; i++) { //看看存不存在这个节点

**if** (p->pNext[i] != NULL) { //存在的才需要管失败指针。

**if** (p == root) { //如果爸爸是根节点的话

p->pNext[i]->Fail = root; //指向根节点

} **else** {

**struct** node \*FailNode = p->Fail; //首先找到爸爸的失败指针

**while** (FailNode != NULL) {

**if** (FailNode->pNext[i] != NULL) { //存在

p->pNext[i]->Fail = FailNode->pNext[i];

**break**;

}

FailNode = FailNode->Fail; //回溯

}

**if** (FailNode == NULL) { //如果还是空，那么就指向根算了

p->pNext[i]->Fail = root;

}

}

que[tail++] = p->pNext[i]; //这个id是存在的，入队bfs

} **else** **if** (p == root) { //变化问题，使得不存在的边也建立起来。

p->pNext[i] = root;

} **else** {

p->pNext[i] = p->Fail->pNext[i]; //变化到LCP。可以快速匹配到病毒。

}

}

head++;

}

**return** ;

}

**int** searchAC(**struct** node \*T, **char** str[]) {

**int** ans = 0;

**struct** node \*p = T;

**struct** node \*root = T;

**if** (p == NULL) **return** 0;

**for** (**int** i = 1; str[i]; i++) { //遍历主串中的每一个字符

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

p = p->pNext[id]; //去到这个节点，虚拟边也建立起来了，所以一定存在。

**struct** node \*temp = p; //p不用动，下次for就是指向这里就OK，temp去找后缀串

//什么叫找后缀串？就是，有单词 she,he 串\*\*\*she，那么匹配到e的时候，she统计成功

//这个时候，就要转移去到he那里，也把he统计进去。**也就是找等价态**

**while** (temp != root && temp->flag != -1) { //root没失败指针

ans += temp->flag;//是单词就加上作为答案

temp->flag = -1; //每个单词只用一次

temp = temp->Fail;

}

}

**return** ans;

}

**char** str[maxn];

**void** work () {

t = 0; //清空字典树，下面的T = NULL也是一样，然后create也清空了。

**struct** node \*T = NULL;

**int** n;

scanf("%d", &n);

**while** (n--) {

scanf("%s", str + 1);

insert(&T, str);

}

scanf("%s", str + 1);

BuiltFail(&T);

printf("%d**\n**", searchAC(T, str));

**return** ;

}

**int** main() {

**int** t;

scanf("%d", &t);

**while** (t--) {

work ();

}

**return** 0;

}

## 8、字符串Hash，找区间不同子串个数

首先介绍一个字符串Hash的优秀映射函数：**BKDRHash**，这里hash一开始是等于0的

for(i=1 to lenstr) hash = seed \* hash + str[i]; 这是求解hash值的公式。多数情况下能唯一确定字符串，seed是一个参数，一般取 31 、131、 1313、 13131、 131313、冲突比较小

经典题目：HDU 4622 Reincarnation

题意：给定一个长为2000个字符串，给出Q(Q<=10000)个询问。每个询问包含[L,R]，要求算出这个区间内不同的子串的个数。

思路：暴力枚举区间长度L，从1开始枚举到lenstr，再枚举起点i即可。能在O(n2)的时间枚举完。但仅仅是枚举完，但这里并没有去重，这部分时间，我们用hash来完成，复杂度压到O(1)。什么叫去重呢？例如baba,当我们枚举第二个ba的时候，就要告诉我们”ba”在[1,4]中重复出现了一次，所以ans[1][4]--; //ans[L][R]就是表示区间内不同子串的个数了。

要枚举那么多子串，我们希望，对于任意给定的区间[L,R]，都能快速地算出它的hash值是多少。例如求[3,4]的hash值，明显有 ans = seed \* str[3] + str[4];（**这是根据公式得到的。**）

那么我们先预处理一个前缀hash总和，记为sumHash[i]表示1~i的hash值。则有

sumHash[1] = str[1]; sunHash[2] = seed \* str[1] + str[2];

sunHash[3] = seed \* sumHash[2] + str[3]; sumHash[4] = seed \* sumHash[3] + str[4];

把他们拆出来，即可得到[3,4] ans = sumHash[4] – seed(R-L+1) \* sumHash[2];

所以预处理两个数组，powseed[i]表示seed的i次方， sumHash[i]定义如上

然后就是怎么判断重复出现的问题了。我们知道那个hash值是唯一的，我们只能靠这个来判断是否重复出现，但是这个hash值很大，用map<ULL,int>来模拟又超时。怎么办呢?我们可以用图，先把hash值%MOD压缩下，把他们加入到一幅图中，再开一个数组保存边的权值，用边的权值来和hash值判断相不相同，即可确定是否重复出现。

#include <cstdio>

#include <cstdlib>

#include <cstring>

**typedef** **unsigned** **long** **long** **int** ULL;

**const** **int** seed = 13131; *// 31 131 1313 13131 131313 etc..*

**const** **int** maxn = 2000 + 20;

**char** str[maxn];

ULL powseed[maxn]; // seed的i次方 爆了也没所谓，sumHash的也爆。用了ULL，爆了也没//所谓，也能唯一确定它，因为是无符号的

ULL sumHash[maxn]; //前缀hash值

**int** ans[maxn][maxn]; //ans[L][R]就代表ans,就是区间[L,R]内不同子串的个数

**const** **int** MOD = 10007;

**struct** StringHash {

**int** first[MOD + 2], num; // 这里因为是%MOD ，所以数组大小注意，不是maxn

ULL EdgeNum[maxn]; // 表明第i条边放的数字(就是sumHash那个数字)

**int** next[maxn], close[maxn]; //close[i]表示与第i条边所放权值相同的开始的最大位置

//就比如baba，现在枚举长度是2，开始的时候ba，close[1] = 1;表明"ba"开始最大位置

//是从1开始，然后枚举到下一个ba的时候，close[1]要变成3，开始位置从3开始了

**void** init () {

num = 0;

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

**return** ;

}

**int** insert(ULL val, **int** id) { //id是用来改变close[]的

**int** u = val % MOD; //这里压缩了下标，val是一个很大的数字，

**for** (**int** i = first[u]; i ; i = next[i])

//存在边不代表出现过，出现过要用val判断，val才是唯一的，边还是压缩后(%MOD)的呢

{

**if** (val == EdgeNum[i]) { //出现过了

**int** t = close[i];

close[i] = id;//更新最大位置

**return** t;

}

}

++num; //没出现过的话，就加入图吧

EdgeNum[num] = val; // 这个才是精确的，只能用这个判断

close[num] = id;

next[num] = first[u];

first[u] = num;

**return** 0;//没出现过

}

} H;

**void** work () {

scanf("%s", str + 1);

**int** lenstr = strlen(str + 1);

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

sumHash[i] = sumHash[i - 1] \* seed + str[i];

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

**for** (**int** L = 1; L <= lenstr; ++L) { //暴力枚举子串长度

H.init();

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

**int** pos = H.insert(sumHash[i + L - 1] - powseed[L] \* sumHash[i - 1], i);

ans[i][i + L - 1] ++; //ans[L][R]++，自己永远是一个

ans[pos][i + L - 1]--; //pos放回0是没用的

//就像bababa，第二个ba的时候，会ans[1][4]--;表明[1,4]重复了一个

//然后第三个ba的时候，ans[2][6]--,同理，表明[2,6]也是重复了

//那么ans[1][6]重复了两个怎么算？就是在递推的时候，将ans[2][6]的值覆盖上来的

//ans[1][6] += ans[2][6] + ans[1][5] - ans[2][5];

}

}

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

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

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

}

}

**int** m;

scanf("%d", &m);

**while** (m--) {

**int** L, R;

scanf("%d%d", &L, &R);

printf("%d**\n**", ans[L][R]);

}

**return** ;

}

**int** main() {

powseed[0] = 1;

**for** (**int** i = 1; i <= maxn - 20; ++i)

powseed[i] = powseed[i - 1] \* seed;

**int** t;

scanf ("%d", &t);

**while** (t--) work();

**return** 0;

}

## 9、Suffix Array

sa[i]表示排名第i为的后缀的起始位置是什么，rank[i]表示第i个字符为起始点的后缀，它的排名是什么。可以知道sa[rank[i]] = i; rank[sa[i]] = i; ★、多case是不需要清空的。

DA算法，倍增算法，复杂度O(nlogn)

//调用函数的时候，在字符串结尾加上一个0，然后参数传递要长度 + 1

da (str, sa, lenstr + 1, mx + 2); // mx的话，如果全是字母，则128够了，注意是lenstr + 1

CalcHight (str, sa, lenstr + 1); //也是 lenstr + 1。就是传到那个0即可

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

str[] = “aabaaaab” + ‘$’; lenstr = 8。**下面的长度都是 lenstr + 1，去到那个0**

rank[] = { 5，7，9，2，3，4，6，8，1} rank[lenstr + 1]必定是1为**无效值**

sa[] = { 9，4，5，6，1，7，2，8，3} sa[1] 必定是lenstr + 1为**无效值**

height[] = { 0，0，3，2，3，1，2，0，1} 前2个是无效值，height[lenstr + 1]是**有效值**

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

**int** sa[maxn], x[maxn], y[maxn], book[**maxn**]; //book[]大小起码是lenstr，book[rank[]]

**bool** cmp(**int** r[], **int** a, **int** b, **int** len) { //这个必须是int r[]，

**return** r[a] == r[b] && r[a + len] == r[b + len];

}

**void** da(**char** str[], **int** sa[], **int** lenstr, **int** mx) {

**int** \*fir = x, \*sec = y, \*ToChange;

**for** (**int** i = 0; i <= mx; ++i) book[i] = 0; //清0

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

fir[i] = str[i]; //开始的rank数组，只保留相对大小即可，开始就是str[]

book[str[i]]++; //统计不同字母的个数

}

**for** (**int** i = 1; i <= mx; ++i) book[i] += book[i - 1]; //统计 <= 这个字母的有多少个元素

**for** (**int** i = lenstr; i >= 1; --i) sa[book[fir[i]]--] = i;

// <=str[i]这个字母的有x个，那么，排第x的就应该是这个i的位置了。

//倒过来排序，是为了确保相同字符的时候，前面的就先在前面出现。

//p是第二个关键字0的个数

**for** (**int** j = 1, p = 1; p <= lenstr; j <<= 1, mx = p) { //字符串长度为j的比较

//现在求第二个关键字，然后合并（合并的时候按第一关键字优先合并）

p = 0;

**for** (**int** i = lenstr - j + 1; i <= lenstr; ++i) sec[++p] = i;

//这些位置，再跳j格就是越界了的，所以第二关键字是0，排在前面

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

**if** (sa[i] > j) //如果排名第i的起始位置在长度j之后

sec[++p] = sa[i] - j;

//减去这个长度j，表明第sa[i] - j这个位置的第二个是从sa[i]处拿的，排名靠前也//正常，因为sa[i]排名是递增的

//sec[]保存的是下标，现在对第一个关键字排序

**for** (**int** i = 0; i <= mx; ++i) book[i] = 0; //清0

**for** (**int** i = 1; i <= lenstr; ++i) book[fir[sec[i]]]++;

**for** (**int** i = 1; i <= mx; ++i) book[i] += book[i - 1];

**for** (**int** i = lenstr; i >= 1; --i) sa[book[fir[sec[i]]]--] = sec[i];

//因为sec[i]才是对应str[]的下标

//现在要把第二关键字的结果，合并到第一关键字那里。同时我需要用到第一关键//字保存的记录，所以用指针交换的方式达到快速交换数组中的值

ToChange = fir, fir = sec, sec = ToChange;

fir[sa[1]] = 0; //固定的是0 因为sa[1]固定是lenstr那个0

p = 2;

**for** (**int** i = 2; i <= lenstr; ++i) //fir是当前的rank值，sec是前一次的rank值

fir[sa[i]] = cmp(sec, sa[i - 1], sa[i], j) ? p - 1 : p++;

}

**return** ;

}

height[i]表示suffix(sa[i])和suffix(sa[i - 1])的LCP，就是两个排名紧挨着的LCP。可知道，sa[1]是最后末尾那个0（因为字典序总是最小的），而它没有前一个后缀，所以height[1] = 0是一定的。同理，sa[2]和sa[1]，也是没有交集的，因为sa[1]的开头就是那个0，所以height[2] = 0也是一定的。而相反，height[lenstr + 1]是有定义的，因为被0占据了sa[1]，所以其他的后移一位。

**int** height[maxn], RANK[maxn];

**void** calcHight(**char** str[], **int** sa[], **int** lenstr) {

**for** (**int** i = 1; i <= lenstr; ++i) RANK[sa[i]] = i; //O(n)处理出rank[]

**int** k = 0;

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

//最后一位不用算，最后一位排名一定是1，然后sa[0]就尴尬了

k -= k > 0;

**int** j = sa[RANK[i] - 1]; //排名在i前一位的那个串，相似度最高

**while** (str[j + k] == str[i + k]) ++k;

height[RANK[i]] = k;

}

**return** ;

}

求任意两个后缀的LCP：

**LCP( suffix(i), suffix(j) ) = min(height[rank[i] + 1] ….…. height[ rank[j] ] )**。

求解sa[7] 和sa[3]的LCP等于中间的sa[4]和sa[3]的LCP、sa[5]和sa[4]、sa[6]和sa[5]、sa[7]和sa[6]的LCP的**最小值**，也就是取公共部分。然后一路传递上去。

**RMQ\_MIN(rank[i], rank[j]); //**这里相对大小不固定，可能要转换一下。 **然后begin++;**

## 杂项

1、回文串的等价定义是：出现奇数次的字母的种类最多只有一种。

2、主串：匹配串。 子串：模式串。

# 数据结构

## 1、并查集

所谓种类并查集，题型一般如下：给定一些基本信息给你，然后又给出一些信息，要求你判断是真是假。例如给出a和b支持不同的队伍，而且b和c也是支持不同的队伍，由于队伍只有两支（就是说只有两种），所以可以推出a和c是支持同一个队伍。

你可能会想用两个并查集，一个并查集存放一个队伍。但是这样是不行的，十分麻烦。因为你想想，如果给出[a,b]不同，然后[c,d]不同，如果我按照左边的放在同一个集合，那么我接着[a,c]不同，这样就会是(a,d)相同，这样的话，你要更改那个并查集，是十分麻烦的。

正解：只用一个并查集，而且再维护一个数组rank[i]表示i与father的关系，0表示支持同一个球队，1表示不同，这样的话，就可以根据rank[x]==rank[y]来判断是不是支持相同的了。爸爸支持谁没所谓啊，我们不关心支持哪个球队，我们只关心支持的是否一样罢了。rank[]数组压缩路径和并查集一样的，只不过其中要列些数据，推些公式出来。

例题：POJ 1733 Parity game

题意：给定一个n(n<=1e10)表示一个长度为n的串，满足如下m(m<=5000)个关系。[a,b]中1的数目为偶数，或者1的数目为奇数。要你判断最多能满足前多少个条件。（条件和以前的相反就直接break）

思路：已知[a,b]中1的数目可以用sum[b]-sum[a-1]表示。由于我们只关心奇偶。如果它说[a,b]是偶数的话，证明sum[b]和sum[a-1]的奇偶性相同。那么这就好办了，如果他说[1,4]是偶数，[3,4]是奇数，那么[1,2]就是偶数了，给出关系[1,2]的时候就可以判断能不能成立了。

这题区间过大(n<=1e10)，但是关系数才5000，所以需要离散化一下就好了。那么问题来了，离散化需要保存相对大小吗？就是如果他说[1,1e10]是偶，那么1e10应该离散一个很大的值，可能是5000了（最多离散也就5000）。这样其实也能做到，把每次输入的排序，再离散。

但是真的要保存相对大小吗？我们想想并查集的时候，我们能决定哪个做大佬的吗？你可能说能啊，我用f[fx]=fy的，就是fy做大佬啊。但是你还要看输入数据的吧。[a,b]和[b,a]，就能把你这个假设搞乱，虽然他这里的区间不能是[10,1]这样，但是意思一样。所以我们不需要保持相对大小，直接离散即可。这里只有一组样例，找到不合适，直接break即可。

#include <cstdio>

#include <cstdlib>

**const** **int** maxn = 5000 + 20;

**int** f[maxn];

**int** rank[maxn];

**void** init () {

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

f[i] = i;

rank[i] = 0; //=0表示奇偶性相同

}

**return**;

}

**int** find (**int** u) {

**if** (u == f[u]) {

**return** u;

} **else** {

**int** temp = f[u]; //记录成只有三个点连成一条线。要记录当前的爸爸是谁，

f[u] = find(f[u]); //不然这里就压缩完了。找不到本来的爸爸了。

rank[u] = (rank[temp] + rank[u]) % 2; //u本来爸爸和它爸爸的关系+我和爸爸的关系

**return** f[u]; //决定了我和新爸爸的关系

}

}

**int** merge (**int** x, **int** y, **int** flag) {

**int** fx = find(x);

**int** fy = find(y);

**if** (fx == fy) { *//有关系了的*

**if** ((rank[x] + rank[y]) % 2 != flag) {

//比如1-->4是奇(rank[1]=1)，2-->4是奇(rank[2]=1)。

//那么你再说1-->2是奇，就GG了 (rank[1]+rank[2])=2，然后%2=0不是1(奇)

**return** 0;

} **else** **return** 1; //成立

} **else** {

f[fx] = fy;

rank[fx] = (rank[x] + rank[y] + flag) % 2; //这个公式有点难推

**return** 1;

}

}

**void** work () {

init();//must be init();

map<**int**, **int**>book; //离散化，不需要安排区间相对大小

**int** n, m, i, liu = 0; //liu是离散化的值

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

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

**int** a, b, flag = 0; //flag=0表示奇偶性相同

**char** str[10];

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

**if** (str[0] == 'o') flag = 1; //奇偶性不同咯

**if** (!book[a - 1]) book[a - 1] = ++liu; //如果还没离散的话。记得是a-1

**if** (!book[b]) book[b] = ++liu; //就给一个离散值他

**if** (!merge(book[a - 1], book[b], flag)) {

**break**;

}

}

printf ("%d**\n**", i - 1);

**return** ;

}

**int** main () {

work ();

**return** 0;

}

关于我说那个难推的公式，可以看着样例推：

10 给出[1,2]，那么我们就，然后[3,4]，[5,6]如此，判定[1,6]就GG了。

5

1 2 even 0 1 0

3 4 odd

5 6 even

1 6 even

7 10 odd 这个是顺向思维，看看逆向的，如果已知[2,4]是奇，那么我再[0,2]是偶，这个时候，0那个点是压缩去4那里的（因为2的父亲是4）那么应该是什么值呢？是1，表明[0,4]这个区间是奇数个1的。这样一路下去，就能推出公式。

变形题目：TOJ3413 How Many Answers Are Wrong

题意：给定一个长为n(n<= 200000)的串，有m(m<= 40000)个关系，告诉你区间[a,b]的和是c，要你找出有多少个关系是和前面的矛盾的。

思路：一样的，注意到[a,b]的和为c可以转化为sum[b]-sum[a-1]，所以目标转化为两个点之差为c。设rank[i]表示其与根节点的差。

#include <cstdio>

#include <cstdlib>

#include <cstring>

**const** **int** maxn = 500000 + 20;

**int** f[maxn];

**int** rank[maxn];

**void** init () {

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

f[i] = i;

rank[i] = 0;

}

**return** ;

}

**int** find (**int** u) {

**if** (f[u] == u) {

**return** u;

} **else** {

**int** temp = f[u];

f[u] = find(f[u]);

rank[u] = (rank[temp] + rank[u]);

**return** f[u];

}

}

**int** merge (**int** a, **int** b, **int** c) {

**int** fa = find(a);

**int** fb = find(b);

**if** (fa == fb) { //rank[a]必须会比rank[b]大 因为a<=b

**if** (rank[a] - rank[b] != c) {

**return** 0;

} **else** **return** 1;

} **else** {

f[fa] = fb;

rank[fa] = (rank[b] - rank[a] + c); //爸爸到新根节点的距离

//就例如有了[5,9]的和为10，,也知道[2,3]的和为1，现在新加入[2,4]的和为7

//这个时候，更改的是[2]的爸爸fa(3)到根节点9的距离，所以，要这样计算

//rank[4]:4到根节点的距离：10，一定要减去rank[2]，2到根节点3的距离1，再+7

//这样得到的才是rank[fa]，就是3到根结点9的距离，是16

**return** 1;

}

}

**int** n, m;

**void** work () {

init ();

**int** ans = 0;

**while** (m--) {

**int** a, b, c;

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

**if** (!merge(a - 1, b, c)) {

ans++;

}

}

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

**return** ;

}

**int** main () {

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

work ();

**return** 0;

}

**并查集的点删除：**HDU 2473 Junk-Mail Filter

关键是怎么分离，可以考虑把它变成一个其它值。HASH[i] = other\_val，然后用新值去做并查集即可、需要注意的一点就是假如现在根是1，fa[1] = 1, fa[2] = 1, fa[3] = 1那么如果你删除了1，这应该输出2.但是现在是fa[2] = 1，是一个不存在的根了，这个时候ans应该+1，但是按照并查集的思路if (find(HASH[i]) == HASH[i]) ans++；是不行的，因为这个根已经不存在了。

解决方法就是标记是否为虚根，del[i] = true表示删除了，但是枚举fa[3]的时候就要避免重新加，需要取消标记。如果这时再有M 4 2，那么就把fa[4] = 1，用虚根表示即可。

**并查集的边删除：**uva 6910 - Cutting Tree

逆序操作，先把残缺的图弄出来，然后把删除操作变成了加边操作。Hack点是如果多次删除了某一条边，那么只能在第一次删除这条边的地方添加这条边进来。

## 2、线段树

多次使用sum不用清0，add要。build的时候就会初始化sum数据。但其他用法就可能要

#define lson L, mid, cur << 1

#define rson mid + 1, R, cur << 1 | 1

**void** pushUp(**int** cur) {

sum[cur] = sum[cur << 1] + sum[cur << 1 | 1];

}

**void** pushDown(**int** cur, **int** total) { //修改的都是儿子的东西，因为**自己的已经加过了**。

**if** (add[cur]) {

add[cur << 1] += add[cur]; //传递去左右孩子

add[cur << 1 | 1] += add[cur]; // val >> 1 相当于 val / 2

sum[cur << 1] += add[cur] \* (total - (total >> 1)); //左孩子有多少个节点

sum[cur << 1 | 1] += add[cur] \* (total >> 1); //一共控制11个，则右孩子有5个

add[cur] = 0;

}

}

**void** build(**int** L, **int** R, **int** cur) {

**if** (L == R) {

sum[cur] = a[L];

**return**;

}

**int** mid = (L + R) >> 1;

build(lson);

build(rson);

pushUp(cur);

}

**void** upDate(**int** begin, **int** end, **int** val, **int** L, **int** R, **int** cur) {

**if** (L >= begin && R <= end) {

add[cur] += val;

sum[cur] += val \* (R - L + 1); //这里加了一次，后面pushDown就只能用add[cur]的

**return**;

}

pushDown(cur, R - L + 1); //这个是必须的，因为下面的pushUp是**直接等于**的

//所以要先把加的，传递去右孩子，然后父亲又调用pushUp，才能保证正确性。

**int** mid = (L + R) >> 1; //一直分解的是大区间，开始时是[1, n]这个区间。

**if** (begin <= mid) upDate(begin, end, val, lson); //只要区间涉及，就必须更新

**if** (end > mid) upDate(begin, end, val, rson);

pushUp(cur);

}

**int** query(**int** begin, **int** end, **int** L, **int** R, **int** cur) {

**if** (L >= begin && R <= end) {

**return** sum[cur];

}

pushDown(cur, R - L + 1);

**int** ans = 0, mid = (L + R) >> 1;

**if** (begin <= mid) ans += query(begin, end, lson); //只要区间涉及，就必须查询

**if** (end > mid) ans += query(begin, end, rson);

**return** ans;

}

## 3、树状数组

**int** c[maxn];//树状数组，多case的记得要清空

**int** lowbit(**int** x) { //得到x二进制末尾0的个数的2次方 2^num

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

}

**void** add(**int** pos, **int** val) { //在第pos位加上val这个值 pos不能是0

**while** (pos <= n) { //n是元素的个数

c[pos] += val;

pos += lowbit(pos);

}

**return** ;

}

**int** get\_sum(**int** pos) { //求解：1--pos的总和

**int** ans = 0;

**while** (pos) {

ans += c[pos];

pos -= lowbit(pos);

}

**return** ans;

}

3.1、求逆序对：首先先把数据离散化，因为树状数组覆盖的区间是1—max这样的，不离散的话开不到那么大的数组。例如离散后是：5、2、1、4、3、思路是：插入5，add(5,1)，把pos为5的地方设置为1，然后ans += i - get\_sum(5); **i的意思是当前插入了i个数**，然后get\_sum()是当前有多少个数比5少，其实就是问1—5之间存在多少个数，那当然是比5小的啦。一减，就是关于5逆序对个数。eg:关于2的逆序对个数是1对。

LL get\_inversion(**int** a[], **int** lena) { //求逆序对个数

LL ans = 0;//逆序对一般都很多，需要用LL

**for** (**int** i = 1; i <= lena; ++i) { // a[]={5,2,1,4,3} ans=6;

// a[]={5,5,5,5,5} ans=0; 逆序对严格大于

add(a[i], 1);

ans += i - get\_sum(a[i]);

}

**return** ans;

}

关于数据离散化，可以开一个结构体，保存val和pos，**然后根据val排序一下**，根据pos从小到大赋值即可。for (int i=1;i<=n;++i) a[book[i].pos]=i; //从小到大离散。a[3]=1,a[1]=2等等

{9,1,0,5,4} 离散化后 {5,2,1,4,3}

3.2、求解区间不同元素个数，离线算法。复杂度O（q + nlog(n)）

设树状数组的意义是：1--pos这个段区间的不同元素的种类数。怎么做？就是add(pos,1);在这个位置中+1，就是说这个位置上元素种类+1。然后先把询问按R递增的顺序排序。因为这里是最优的，我每次尽量往R靠，使得查询不重不漏。什么意思呢？就是假如有：2、1、3、5、1、7的话。一开始的[1,4]这段数字全部压进树状数组，用个数组book[val]，表示val这个元素出现的最右的位置，因为我们需要删除重复的，也是要尽量往右靠。到达pos=5这个位置的时候，注意了，因为1是出现过的book[1] = 2，所以我们要做的是把2这个位置出现元素的种类数-1，就是add(book[1], -1)。然后把第五个位置出现的元素种类数+1，就是add(5,1)。为什么呢？因为你尽量把种类往右靠，因为我们的R是递增的，这样，你使得查询[4,6]成为可能，因为我那个1加入来了，而不是一直用pos=2那个位置的1，再者，查询[4,7]的话，一样的意思，因为中间的1进来了。所以我们因为尽量往右靠，毕竟我们都把query按R排序了。还有这个只能离线，一直预处理ans[i]表示第i个询问的ans。更新到[4,7]后，查询[1,2]已经不可能了，因为很明显，pos=2这个位置已经被删除了。

**离散化只是为了book[val]成为可能**

**void** work() {

scanf("%d", &n);

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

**int** q;

scanf("%d", &q);

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

scanf("%d%d", &query[i].L, &query[i].R);

query[i].id = i; //记录ans

}

sort(query + 1, query + 1 + q);

**int** cur = 1;

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

**for** (**int** j = cur; j <= query[i].R; ++j) {

**if** (book[a[j]])

add(book[a[j]], -1); //del 这个位置

book[a[j]] = j; //更新这个位置的最右值

add(j, 1); //这个位置出现了新元素

}

cur = query[i].R + 1; //表示现在预处理到这个位置了。不能往回查，而且也不会往回

ans[query[i].id] = get\_sum(query[i].R) - get\_sum(query[i].L - 1); //区间减法

}

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

printf ("%d**\n**", ans[i]);

}

二维树状数组：就是n个一维树状数组，然后，在这些一维树状数组里面，再套一个一维树状数组。也就是一维树状数组的树状数组。

所以c[1][1]维护的是第一行的树状数组的前1个，c[1][2]是第一行树状数组的前2个，也就是a[1][1] + a[1][2]。这个和一维bit的定义是一样的。

然后c[2][1]维护的是第一行的bit + 第二行的bit的前1个、也就是a[1][1] + a[2][1]，

c[2][2]维护的是a[1][1] + a[1][2] + a[2][1] + a[2][2]。

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

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

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

c[i][j] += val;

}

}

}

**int** sum(**int** x, **int** y) {

**int** ans = 0;

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

**for** (**int** j = y; j >= 1; j -= lowbit(j)) {

ans += c[i][j];

}

}

**return** ans;

}

## 4、RMQ

dp\_min[i][j] 表示区间[i,i+2^j-1]的最小值。因为这个区间长度是偶数，易得转移方程是把区间分成两半再合并。然后查询的时候应为要精确覆盖到区间[begin,end]，不能多，也不能少。那么最好的办法就是考虑区间重叠了。例如：[2,7]可以考虑[2,5]和[4,7]的最小值。假如我们需要查询的区间为(i,j)，那么我们需要找到覆盖这个闭区间(左边界取i，右边界取j)的最小幂

预处理复杂度O(nlogn)，然后可以O(1)查询 。 NYOJ 119

**void** init\_RMQ(**int** n, **int** a[]) { //预处理->O(nlogn)

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

dp\_max[i][0] = a[i]; //自己

dp\_min[i][0] = a[i]; //dp的初始化

}

**for** (**int** j = 1; j < 20; ++j) { //先循环j，不取等号，220是1e6了

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

dp\_max[i][j] = max(dp\_max[i][j - 1], dp\_max[i + (1 << (j - 1))][j - 1]);

dp\_min[i][j] = min(dp\_min[i][j - 1], dp\_min[i + (1 << (j - 1))][j - 1]);

}

}

**return** ;

}

**int** RMQ\_MAX(**int** begin, **int** end) {

**int** k = (**int**)log2(end - begin + 1.0);

**return** max(dp\_max[begin][k], dp\_max[end - (1 << k) + 1][k]);

}

**int** RMQ\_MIN(**int** begin, **int** end) {

**int** k = (**int**)log2(end - begin + 1.0);

**return** min(dp\_min[begin][k], dp\_min[end - (1 << k) + 1][k]);

}

## 5、单调队列 & 单调栈

单调队列：给定数组a，n <= 1e5。多次任取其中连续的k个数字，求区间元素的最大值。思路就是保存一个单调递减的队列，因为其元素可能会失效，所以也要保存一个id记录的是队列元素在a[]中的位置，如果它是 <= i – k 的，那么就是已经失效的了，应该head++

que[1].val = a[1]; que[1].id = 1;

**int** head = 1, tail = 1;

**for** (**int** i = 2; i <= k; ++i) { //开始先预处理前k个。

**while** (tail >= head && a[i] >= que[tail].val) --tail;

++tail;

que[tail].val = a[i]; que[tail].id = i;

}

ans\_max[++lenmax] = que[head].val;

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

**while** (tail >= head && a[i] >= que[tail].val) --tail; //这里等于也进来，因为是最新的。

++tail;

que[tail].val = a[i]; que[tail].id = i;

**while** (head <= tail && que[head].id <= i - k) ++head; //把废弃了的排除掉

ans\_max[++lenmax] = que[head].val; //每次出队就是最大值了。

}

单调栈：给定数组a，区间a[L…R]的价值是最小元素乘上区间总和。求出最大价值。

首先用个单调递增的栈，同时维护一个lef[i]表示当前栈中第i个元素的最左区间，就是再左一点就不是它最小了。那么可以知道stack[i]在区间[left[i], 栈顶元素位置]，这个区间是最小的。因为本来栈顶元素就是最大的了，栈内元素绝对不够它大。然后每次弹出一个元素，就能知道它在区间内最小，就能计算了。每弹出一个元素，都计算一次。

**for** (**int** i = 1; i <= n + 1; ++i) { // 加个a[n + 1] = -1 防止全部单调递增。

**int** TT = stack[top]; // 栈内元素到栈顶元素，这个区间最小值

**int** toleft = i; //保存插入后上一个元素的最左值

**while** (top >= 1 && a[i] < a[stack[top]]) { //等于的话，不如让它扩大

LL t = (sum[TT] - sum[lef[stack[top]] - 1]) \* a[stack[top]];

**if** (t > ans) {

ans = t; L = lef[stack[top]]; R = TT;

}

toleft = lef[stack[top]]; //保留栈内元素最左值，更新这个元素的最左值

--top;

}

++top; stack[top] = i; lef[stack[top]] = toleft;

}

## 6、分块

动态RMQ问题。其实分块后自己想**块内**维护什么就慢慢想吧。**Note:数组必须从0开始**

**void** init() { // O(n)预处理

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

**if** (i % Magic == 0 || a[i] > mx[i / Magic]) {

mx[i / Magic] = a[i];

}

}

**return** ;

}

**void** update(**int** pos, **int** val) { // O(sqrt(n)) 修改

a[pos] = val; //更改

**int** L = pos / Magic \* Magic; //算出在第几个块的左值

**int** R = L + Magic - 1; *//右*

**for** (**int** i = L; i <= R; ++i) {

**if** (i % Magic == 0 || a[i] > mx[i / Magic]) {

mx[i / Magic] = a[i];

}

}

**return** ;

}

**int** query(**int** begin, **int** end) { //O(sqrt(n)) 查询

**int** ans = -inf;

**for** (**int** i = begin; i <= end;) {

**if** (i % Magic == 0 && i + Magic - 1 <= end) { //前后两段要暴力

ans = max (mx[i / Magic], ans);

i += Magic;

} **else** {

ans = max (a[i], ans);

++i;

}

}

**return** ans;

}

## 杂项

1、区间等差数列更新和单点查询

就是根据打标记的思想来做，但是公差可能会重叠，所以另开一个数组保存，原数组就一个关键的地方，计算出末尾项是什么，这样才能结束更新，不影响后面的值。

**void** addL(**int** begin, **int** end, **int** val, **int** d) { //在区间上加上首项为val，公差为d的数列

cntL[begin] += val;

cntL[end + 1] -= d \* (end - begin) + val; //计算出末尾值，否则影响后面的值。

subL[begin + 1] += d; //首项不用减去公差，所以在begin+1上开始

subL[end + 1] -= d; //结束公差

**return** ;

}

**void** init\_arr(**int** end) { //更新即可。

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

subL[i] += subL[i - 1]; //把公差传递下去，先计算公差，后算值。

cntL[i] += cntL[i - 1] + subL[i]; //a[i] = a[i-1] + d 等差数列公式。

}

**return** ;

}

# 动态规划

## 1、整数划分

4 = 4; 4 = 3 + 1; 4 = 2 + 2; 4 = 2 + 1 + 1; 4 = 1 + 1 + 1 + 1;

方法一：dp[i][j]表示拆分数为i的时候，最小拆分数是j的时候的解。（前缀和，包括最小拆分数是其他数字的解。所以dp[val][1]就是答案。）**dp[i][j] = dp[i][j + 1] + dp[i - j][j];** 如果需要不重复数字，则是dp[i – j][j + 1]; 因为这样限定了最小拆分数一定是唯一的一个j。如果要输出最小拆分数固定是k时的解，dp[val][k] – dp[val][k + 1]; 边界：**dp[i][i] = 1;**

方法二：设dp[i][j]表示拆分数为j的时候，拆分成i项时的解。**边界：dp[0][0] = 1;**

允许重复数字 ： dp[i][j] = dp[i][j - i] + dp[i - 1][j - 1]; 此时dp[i][i] = 1;

不允许重复 ： dp[i][j] = dp[i][j - i] + dp[i - 1][j - i]; 此时dp[i][i] = 0;

dp[i][j]表示j这个数字，当前的拆分拥有i个拆分数时的方案数。至于为什么要在第二维中放j这个数字，而不是像上面那个用第一维放数字，那是因为它需要先解出dp[1][1---n]然后再解出dp[2][1---n]，转换维度比较方便。

先考虑允许重复数字 ： dp[i][j] = dp[i][j - i] + dp[i - 1][j - 1];

考虑分成两类，

1、dp[i][j - i]：这种拆分方案（拥有i个数字的拆分方案），如果没有1，就比如7 = 3 + 4这样，然后每个数字都加上一，

就变成了9 = 5 + 4。所以dp[2][9]可以由dp[2][7]转化过来。当然7 = 1 + 6也是合法解。

2、dp[i - 1][j - 1]：这种拆分方案有1，比如4 = 3 + 1，那么我可以截去那个1，变成3 = 3，然后加上最后那个1，就变成了4 = 3 + 1，所以dp[2][4]可以由dp[1][3]转化过来。

但是题目需要不重复，（这也使得题目不会超时）

第一类，如果dp[2][7]本来就是不重复的，就是dp[2][6]即是6 = 3 + 3不能发生，那么我同时全部加上一个数，肯定不会产生重复的。

第二类，如果本来也是不重复的，但是生成的可能会重复，比如5 = 4 + 1和5 = 3 + 2是dp[2][5]的解（本来没有重复），然后在后面加上一个1，是dp[3][6]的解，但是6 = 4 + 1 + 1是非法的。我们也不可能检查其拆分方案有没1，因为我们只会统计数量。

改进：dp[i][j] = dp[i][j - i] + dp[i - 1]**[j - i]**;

dp[i - 1][j - i]：意思就是每个元素减去1后，分成i - 1组，为什么不是分成i组呢。？因为其存在1，然后每个数字减去1,那么这个1就是变成0了，所以只能分成i - 1组。

例如：6 = 3 + 2 + 1是由3 = 2 + 1弄过来的，dp[3][6] = dp[2][3]

所以就能解决这题。

1、i > j，不用管dp[i][j] = 0

2、i == j，只能够是i个1,然而不符合题目，dp[i][j] = 0;

3、i < j，dp[i][j] = dp[i][j - i] + dp[i - 1][j - i];

由于之和上一维有关，可以滚动数组，就能解决空间问题。

然后因为要产生k个不同的数字，最小值是1 + 2 + 3 + ..... + k，就是(1 + k) \* k / 2开始。

就是dp[2][2]这些不用枚举了，k = 2的话，最小值是3

## 2、区间DP

dp[i][j]表示把第[i, j]这个区间的石子合并到一堆时所需的最小费用。无论在哪里合并，这一步的操作的费用总是[i, j]这个区间式子的数字总和。暴力枚举切点，然后记忆化搜索。

**int** dfs(**int** be, **int** en) { //开始的时候dp的所有值都是inf

**if** (be >= en) **return** 0;

**if** (vis[be][en] == DFN) **return** dp[be][en];

vis[be][en] = DFN;

**for** (**int** i = be; i <= en; ++i) {

dp[be][i] = dfs(be, i);

dp[i + 1][en] = dfs(i + 1, en);

dp[be][en] = min(dp[be][i] + dp[i + 1][en] + sum[en] - sum[be - 1], dp[be][en]);

}

**return** dp[be][en];

}

## 3、数位DP

HDU 2089 不要62

**int** dp[10][20]; // dp[i][j]表示长度是i的数字中，以j开头的合法情况

**void** init() {

dp[0][0] = 1;

**for** (**int** i = 1; i <= 7; ++i) { //枚举数字的长度是多少位

**for** (**int** j = 0; j <= 9; ++j) { //枚举长度是i的开头数字

**if** (j == 4) **continue**; //4是不合法情况，跳过了，所以dp[i][4] = 0;

**for** (**int** k = 0; k <= 9; ++k) { //枚举长度是i - 1的开头数字

**if** (k == 4 || j == 6 && k == 2) **continue**; //其实这个k = 4不跳过也一样，= 0

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

}

}

}

//dp[i][0] += dp[i - 1][0] + dp[i - 1][1] ... + dp[i - 1][9];

}

注意的是dp[i][0]的意义，dp[i][0]表示长度是i的，以0开头的合法数字。也就是0XX的形式，所以是包含了000---099，也就是所有合法的1位数和2位数之和。那000是什么？不管了，这个是多余的，也可以看作是合法的一位数，虽然题目的区间不包含000，但是R – L的时候会同时抵消这个多余的计数。

比如我统计的是数字: 456

那么，从高到低枚举。先枚举第一位，4、再枚举5、....那么，< 4 的，长度是3位的合法数字，都应该算做贡献。就是ans += dp[3][0...3]，dp[3][1]好理解，就是100、110、....199那些。

dp[3][2]那些同理，不处理到dp[3][4]也好理解，因为dp[3][4]包含了499那些，就是超越范围了。而后来的枚举下一位的5，dp[2][0...4]是同理的，但是其是和第一位的4结合的，也就是dp[2][0]包括了所有1位数的合法情况，然后组合成40X。所以后面的枚举其实这是为了和上一位组合成合法情况。所以当其中某些数字破坏了条件，例如出现了4，或者已经出现了62，就要提前break。还有需要注意的是处理不到n这个数字的，因为都是枚举每一位的-1那个大小，所以只需要把他们 + 1即可。

**int** calc(**int** val) { //calc(R + 1) - calc(L + 1 - 1)

**int** lenstr = 0;

**while** (val / 10 > 0) {

str[++lenstr] = val % 10 + '0';

val /= 10;

}

str[++lenstr] = val + '0';

str[lenstr + 1] = '\0';

**int** ans = 0;

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

**for** (**int** j = 0; j <= str[i] - '0' - 1; ++j) {

**if** (j == 2 && str[i + 1] == '6') **continue**; //62X是不合法的，要跳过

//那么j = 4呢？不跳过？其实是一样的，因为预处理的时候dp[i]][4] = 0的。

ans += dp[i][j];

}

//后来枚举的，都是和前面的哪一位结合，比如456，就是4XX，然后45X

**if** (str[i] == '4') **break**; //4XX，下一次枚举就不合法了

**if** (str[i] == '2' && str[i + 1] == '6') **break**;

}

**return** ans;

}

2、求区间包含“13”这个子串 && 能整除13的个数

设dp[i][j][state][r]表示位数是i，以j开头，state是0或1表示是否已经出现了”13”这个子串，并且，整个数字mod 13 = r的时候的合法个数。统计方法和不要62是一样的

**void** init() {

base[0] = 1;

**for** (**int** i = 1; i <= 10; ++i) {

base[i] = base[i - 1] \* 10; // 10i

}

**for** (**int** i = 0; i <= 9; ++i) { //dp[1][0][0][0] = 1

dp[1][i][0][i] = 1;

}

**for** (**int** i = 2; i <= 10; ++i) { //枚举位数i

**for** (**int** j = 0; j <= 9; ++j) { //枚举第i位的开头数字j

**for** (**int** x = 0; x <= 9; ++x) { //第i - 1位的开头数字x

**int** t = base[i - 1] \* j % 13; //2X中的20 % 13是几

**for** (**int** r = 0; r <= 12; ++r) { //枚举i - 1位的余数

dp[i][j][1][(r + t) % 13] += dp[i - 1][x][1][r];

**if** (j == 1 && x == 3) { //出现了13这个子串*。*

dp[i][j][1][(r + t) % 13] += dp[i - 1][x][0][r];

} **else** {

dp[i][j][0][(r + t) % 13] += dp[i - 1][x][0][r];

}

}

}

}

}

}

**int** calc(**int** n) {

n++;

**int** digit[15] = {0};

**int** lenstr = 0;

**while** (n / 10 > 0) {

digit[++lenstr] = n % 10;

n /= 10;

}

digit[++lenstr] = n % 10;

**int** ans = 0;

**int** mod = 0;

**bool** flag = **false**;

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

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

ans += dp[i][j][1][(13 - mod) % 13];

**if** (flag || j == 3 && digit[i + 1] == 1) {

ans += dp[i][j][0][(13 - mod) % 13];

}

}

**if** (digit[i] == 3 && digit[i + 1] == 1) {

flag = **true**; //前面位的数字已经包含了13、

}

mod = (mod + digit[i] \* base[i - 1]) % 13; //更新5XX中500 % 13的余数，54X中540%13

}

**return** ans;

}

## 4、状压DP

1、合法括号序列的状压，不需要dp[lef][rig]表示左括号lef个，右括号rig个，可以用他们的差值来表示，dp[dis]表示左括号减去右括号的差值。差值小于0是没可能的，因为合法的括号序列不可能出现rig > lef的情况。

2、判断一个数二进制是否存在相邻两位同时为1，只需：(x & (x << 1)) > 0

## 杂项

1、基本背包枚举状态是否成立。（POJ 1948，枚举组合成三角形）

转移可以很直接，枚举新进来的物品，如果dp[i][j]成立，那么dp[i + val][j]也成立。

也可以设dp[i][j]表示用了i个数，能否生成j。dp[i][j] = dp[i][j] || dp[i – 1][j – a[i]];

2、bitset优化的背包，dp[0]表示能否产生这个数字，然后dp = dp | (dp << x)，就比如是0001的时候，枚举了一个3进来，那么就是0001 | 1000 = newState : 1001

3、背包记录路径 （vijos [P1071 新年趣事之打牌](https://vijos.org/p/1071) && POJ 1015）

很多时候是不行的，因为更新了12的最优解，如果它依赖于6这个背包，然后你后面改变了6这个背包，就GG。如果能使得你更新过了的背包，不更新的话，那就可以。**就是只看背包能否成立**，而**不看**其他权值总和最大化的**最优解**的情况下，是可以的。**ie: dp[val] = true**

1、一个优秀的背包记录路径的方法就是，转移的时候判断是否存在于路径之中即可。但是求解dp的时候枚举顺序要发生变化，比如选出m个数的话，要先枚举m，然后枚举n个数，也就是先解出选出1个数的时候（在n个数里面挑）的最优解，然后解选两个数的最优解，但是在解选两个数的时候的最优解的时候，要判断是否存在于路径之中。**判断上一个的路径。**

2、用二维01背包，就可以在能求最优解的情况下同时记录选了哪些数字。

4、LICS、O（lena \* lenb）

设dp[i][j]表示匹配到a[]的前i项，以b[]的第j项结尾时，能匹配的最大值。dp[all][all] = 0;

①、不匹配a[i]这个数，则是dp[i][j] = dp[i – 1][j]; //一定要以b[j]结尾。

②、匹配a[i]这个数，则需要a[i] == b[j] && b[j] > b[k] 🡪 dp[i][j] = max(dp[i – 1][k]) + 1，

这样复杂度需要O(n3)，注意到，求解dp的时候，是从dp[i][1….lenb]这样的顺序求解，而且，需要a[i] == b[j]才能算做贡献，因为要LCS嘛！那么可以记录dp[i][1…j – 1]的信息，以a[i]作为基准（因为a[i] == b[j]才能算出贡献，以那个作为基准没所谓），找出前j - 1个数中，满足LIS并且最大的那个，O(1)更新即可。  
**for** (**int** i = 1; i <= lena; ++i) {

**for** (**int** j = 1, cnt = 0; j <= lenb; ++j) {

dp[i][j] = dp[i - 1][j]; //不要当前这个a[i]

**if** (a[i] > b[j]) { //形成LIS

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

}

**if** (a[i] == b[j]) { //形成LCS

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

}

}

}

ans = max(ans, dp[lena][1…lenb]);

# 图论

设：顶点数为：n 边数为：m

无向完全图有： 条边。 有向完全图，有n(n-1)条边。

存储方式：

1、邻接矩阵：e[maxn][maxn]。对于遍历每一条边，复杂度是O(n²)，适合用于稠密图

2、邻接链表:

用first[u]表示第u个顶点的第一条边是谁。然后用个next[i]表示，第i条边的下一条边是谁，就能完成图的遍历。插入first[u]的时候，用头插法即可。

next[i]=first[u]; first[u]=i;//头插法

C语言写法：

**struct** edge {

**int** u, v, w;

**int** id;//区分无向图的时候用的，id相同就是同一条边

**int** tonext;

} e[maxn \* 2]; //这个存的是边，要插入两次，所以要 \* 2

**int** first[maxn]; //这个表示什么【顶点】的第一条边，所以只用maxn大小即可

**int** num = 0; //从1开始，这样就是没0号这条边。方便判断。所以每次先++num再放边！！

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

++num; //这个num是边的编号

e[num].u = u, e[num].v = v, e[num].w = w;

e[num].tonext = first[u]; //下一条边是这个顶点的第一条边

first[u] = num; //这个顶点的第一条边是编号为num的这条

}

遍历的时候，如果想对u顶点的所有边进行遍历，就只需这样

**for** (**int** i = first[u]; i != 0; i = e[i].next) 现在的i就是边的编号，这条边就是e[i].u -🡪 e[i].v 的边

C++写法：

**struct** edge {

**int** u, v, w;

};

vector<**struct** edge>e[maxn];

加边的时候，例如5-🡪6有一条权值为20的边，那么，开个变量struct edge t; //用于插入

t.u=5; t.v=6; t.w=20; e[t.u].push\_back(t); 如果是无向图，同样还需再add一次。

如果对顶点u遍历： for (int i=0;i<e[u].size();i++) 这样：就是e[u][i].u 🡪 e[u][i].v这条边了

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

**struct** edge t;

t.u = u, t.v = v, t.w = w;

e[u].push\_back(t);

**return** ;

}

多叉树转二叉树，开始的时候，memset为0或者-1都可以。

Lchild[cur] 表示cur这个节点的儿子。

Rchild[cur] 表示cur这个节点的兄弟。

**void** addEdge(**int** u, **int** v) {

Rchild[v] = Lchild[u];

Lchild[u] = v;

}

## 1、最短路径

**★：**最短路的算法都可以用于­­­­---**有向图 & 无向图**

**★：**最短路肯定是一个只包含(n-1)条边的简单路径。

(1)、Dijkstra单源最短路

传入邻接矩阵e[][MAXN]，节点个数n(编号从1--n),最短路径数组dis,和出发点cur，返回inf代表原图不连通，else返回0。复杂度O(n²)。注意路径必须都是非负的。

**一开始的e[][]需要全部设置为inf，**pre[i]表示到达i顶点的上一个顶点，记录路径

**int** dij(**int** e[][maxn], **int** n, **int** dis[], **int** cur, **int** pre[]) {

**bool** book[maxn]= {0}; //记得初始化为0

**for** (**int** i = 1; i <= n; ++i) dis[i] = inf; //只能这样初始化，传参的话，不然sizeof (dis) = 4

dis[cur] = 0;

pre[cur] = -inf;

**for** (**int** i = 1; i <= n; i++) { //最多使用n个点来中转，

**int** mi = inf; //每次找出最小值

**int** u = inf; //下标

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

**if** (!book[j] && mi > dis[j]) {

mi = dis[j];

u = j;

}

}

**if** (u == inf) **return** inf; //原图不连通

book[u] = **true**;

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

**if** (!book[j] && dis[j] > dis[u] + e[u][j]) {

dis[j] = dis[u] + e[u][j]; //进行松弛操作

pre[j] = u;

}

}

}

**return** 0;

}

优先队列优化的dij，复杂度O(M+N)logN，传入起点bx，即可求出dis[]

注意：**M最大是N2，这个时候比O(n2)算法还要慢。**

**struct** HeapNode {

**int** u,dis; //dis是到起始点bx的距离

HeapNode(**int** from, **int** cost) : u(from), dis(cost) {}

**bool** **operator** < (**const** HeapNode &rhs) **const** {

**return** dis > rhs.dis; //注意，这里的dis小的在前。

}

};

**void** dij(**int** bx) {

memset (book, 0, **sizeof** book); // 这些数组只能放在外面，

memset (dis, 0x3f, **sizeof** (dis)); //不然如果是函数传参的话：sizeof (dis) = 4

dis[bx] = 0;

priority\_queue<HeapNode> que;

que.push(HeapNode(bx, dis[bx]));

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

HeapNode t = que.top();

que.pop();

**int** u = t.u; //现在选出的这个u，是dis[]中最小的那个值

**if** (book[u]) **continue**;

book[u] = **true**;

**for** (**int** i = first[u]; i; i = e[i].next) {

**int** v = e[i].v;

**if** (!book[v] && dis[v] > dis[u] + e[i].w) { //找过的点再用也不行的

dis[v] = dis[u] + e[i].w; //松弛

pre[v] = u; //记录路径

que.push(HeapNode(v, dis[v]));

}

}

}

**return** ;

}

(2)、Bellman\_Ford 解决负权边 + 判负环，复杂度O(n \* m)

**int** Bellman\_Ford(**int** bx, **int** n, **int** m) { //从bx开始，有n个点，m条边

dis[bx] = 0;

**for** (**int** i = 1; i <= n - 1; ++i) { //n个点的话，只需n-1条边来松弛，因为是最短路径。

**bool** flag = 0;

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

**if** (dis[e[j].v] > dis[e[j].u] + e[j].w) {

flag = 1;

dis[e[j].v] = dis[e[j].u] + e[j].w;

}

}

**if** (!flag) **break**; //不改变了，就可以提前出来

}

**for** (**int** j = 1; j <= m; ++j) { /\*判负环的话，只需再进行一次\*/

**if** (dis[e[j].v] > dis[e[j].u] + e[j].w) **return** 1; //继续松弛，错误。出现了负环。

}

**return** 0;

}

(3)、SPFA Bellman\_Ford的队列优化，复杂度O(2 \* m)，最坏O(n \* m)

如果一个节点进队n次，那么就是出现了负环。不是被更新n次，因为有可能是1🡪2有10条边，这些边权值重大到小，更新了10次，但是这不是负环。

**bool** spfa(**int** bx, **int** n) { //从bx开始，有n个顶点

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

dis[i] = inf;

tim[i] = 0; //入队次数清0

in[i] = **false**; //当前这个节点不在队列里

}

queue<**int**> que;

**while** (!que.empty()) que.pop();

que.push(bx), in[bx] = **true**, dis[bx] = 0, tim[bx]++;

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

**int** u = que.front();

**if** (tim[u] > n) **return** **true**; //入队次数超过n次，出现负环

que.pop();

**for** (**int** i = first[u]; i; i = e[i].tonext) {

**if** (dis[e[i].v] > dis[e[i].u] + e[i].w) {

dis[e[i].v] = dis[e[i].u] + e[i].w;

**if** (!in[e[i].v]) { //不在队列

que.push(e[i].v);

in[e[i].v] = **true**;

tim[e[i].v]++;

}

}

}

in[u] = **false**;

}

**return** **false**;

}

(4)、floyd的bitset优化，只能用于判断是否到达，复杂度O(n3 / sizeof bitset)

如果是1000位的bool[]，你算a ^ b需要的时间是O(n)，但是如果用bitset直接做，复杂度是O(n / sizeof bitset)。bitset内存是，8bit是一字节，那么长度 / 8就是字节数。

**for** (**int** k = 1; k <= n; ++k) { //枚举点k来中转

**for** (**int** i = 1; i <= n; ++i) { //每次都更新这n个点。

**if** (e[i][k]) { //有点dp的思想，

e[i] |= e[k]; //i能到k的话，就能到达k能到达的所有点。

}

}

}

Floyd算最短路。

**for** (**int** i = 1; i <= n; ++i) { //枚举点i来中转，记得设置e[i][i] = 0

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

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

**if** (e[k][j] > e[k][i] + e[i][j]) e[k][j] = e[k][i] + e[i][j];

}

}

}

## 2、最小生成树（MST）

Minimum Spanning Tree

1、kruskal算法（克鲁斯卡尔算法）

把边从小到大排序，每次选取可行的最小的边，判断可行性用并查集维护，防止连通成图。然后选够了n – 1条边后可以直接break了。复杂度O(ElogE)，主要时间用在了快排那里。

**int** kruskal(**int** n, **int** m) { //n个顶点，m条边

sort(e + 1, e + 1 + m); //最大生成树的话，从大到小排即可

**for** (**int** i = 1; i <= n; ++i) fa[i] = i; //并查集初始化

**int** cnt = 0, ans = 0;

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

**if** (tofind(e[i].u) == tofind(e[i].v)) **continue**; //防止连通成图

tomerge(e[i].u, e[i].v);

cnt++;

ans += e[i].w;

**if** (cnt == n - 1) **return** ans;

}

**return** -1; //原图不联通

}

2、Prim算法（普里姆算法）

记dis[i]表示i号顶点到生成树的距离。一开始，dis[1] = 0表示1号顶点作为生成树的根。然后枚举1号顶点连接到的边，更新其他顶点到现有生成树顶点的距离。所以就是分为生成树顶点和非生成树顶点两类。每一次找出生成树顶点中所有边最小的，去更新非生成树顶点。

普通的复杂度需要O（n2）

堆优化的复杂度MlogN

**int** Prim(**int** n, **int** m) { //n个顶点，m条边

**while** (!que.empty()) que.pop(); //清空优先队列

++DFN;

**for** (**int** i = first[1]; i; i = e[i].tonext) {

que.push(HeapNode(e[i].v, e[i].w));

}

book[1] = DFN; //表明这个点已经加入MST

**int** cnt = 1, ans = 0;

**while** (cnt < n) { //选出n个点出来

**bool** flag = **false**;

**struct** HeapNode t(0, 0);

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

t = que.top(); que.pop();

**if** (book[t.v] == DFN) **continue**;

flag = **true**;

**break**;

}

**if** (!flag) **break**; //原图不连通

book[t.v] = DFN, cnt++, ans += t.val;

**for** (**int** i = first[t.v]; i; i = e[i].tonext) {

**if** (book[e[i].v] == DFN) **continue**; //已经在生成树顶点中了。

que.push(HeapNode(e[i].v, e[i].w));

}

}

**if** (cnt != n) **return** -1;

**else** **return** ans;

}

## 3、LCA、树的重心、树的直径

LCA是用在树中的，**图的不行**。★、无向树也是树。

LCA[u][v]表示节点u和v的最近公共祖先，也是深度最大祖先，深度越大的话，证明离u和v越近嘛。这个算法基于dfs的回溯和并查集实现。dfs的时候，搜索到叶子节点（没有儿子）的时候，得到LCA[u][u]=u和LCA[u][fa]=fa，然后，返回到他爸爸那里，并查集合并，f[u]=fa;表明u的爸爸是fa，所以这个时候并查集是**向左看齐的**，merge(u,v)，u只能是爸爸。复杂度O(n²)的算法，能求出整棵树的所有LCA[i][j]值。

★并查集那里有点奇葩，它也用作了标记数组的作用，所以一开始的并查集，全部是0。

**void** dfs(**int** u) {

f[u] = u; //首先自己是一个集合

**for** (**int** i = first[u]; i; i = e[i].next) {

**int** v = e[i].v;

**if** (f[v] == 0) {

dfs(v);

merge(u, v);

}

}

**for** (**int** i = 1; i <= n; i++) { //遍历每一个点

**if** (f[i]) { //已经确定过的，就更新LCA

LCA[u][i] = LCA[i][u] = find(i);

}

}

**return** ;

}

O(n + Q)算法，用邻接表存取所有询问，要询问的再处理即可，注意去重操作。

**void** dfs(**int** u) {

f[u] = u; //首先自己是一个集合

**for** (**int** i = first[u]; i; i = e[i].next) {

**int** v = e[i].v;

**if** (f[v] == 0) {

dfs(v);

merge(u, v);

}

}

**for** (**int** i = first\_query[u]; i; i = query[i].next) {

**int** v = query[i].v;

**if** (f[v]) { //确定过的话，并且有要求查询

//要求查询的话这个query保存着，first\_query[u]就证明有没了

//因为插边插了两次，这里要去重。用id保存答案即可

ans[query[i].id] = find(v);

}

}

**return** ;

}

LCA倍增算法。

设ansc[cur][i]表示从cur这个节点跳2i步到达的祖先是谁。记录深度数组deep[cur]。深度从0开始，然后算LCA的时候就先把他们弄到同一深度，然后一起倍增。

Hint：ans[root][3]是自己，都是root。开始的时候fa[root] = root。deep[root] = 0;

一般这课树是双向的，因为可能结合bfs来做题，所以需要判断不能走到爸爸那里。

1 << 20就有1048576（1e6）了。

**int** ansc[maxn][25], deep[maxn], fa[maxn];

**void** init\_LCA(**int** cur) {

ansc[cur][0] = fa[cur]; //跳1步，那么祖先就是爸爸

**for** (**int** i = 1; i <= 24; ++i) { //倍增思路，递归处理

ansc[cur][i] = ansc[ansc[cur][i - 1]][i - 1];

}

**for** (**int** i = first[cur]; i; i = e[i].tonext) {

**int** v = e[i].v;

**if** (v == fa[cur]) **continue**;

fa[v] = cur;

deep[v] = deep[cur] + 1;

init\_LCA(v);

}

}

**int** LCA(**int** x, **int** y) {

**if** (deep[x] < deep[y]) swap(x, y); //需要x是最深的

**for** (**int** i = 24; i >= 0; --i) { //从大到小枚举，因为小的更灵活

**if** (deep[ansc[x][i]] >= deep[y]) { //深度相同，走进去就对了。就是要去到相等。

x = ansc[x][i];

}

}

**if** (x == y) **return** x;

**for** (**int** i = 24; i >= 0; --i) {

**if** (ansc[x][i] != ansc[y][i]) { //走到第一个不等的地方，

x = ansc[x][i];

y = ansc[y][i];

}

}

**return** ansc[x][0]; //再跳一步就是答案

}

**树的重心**：求以cur为根的子树的重心，就是要找一个点，使得删除这个点后，分开来的零散的子树中，节点数的最大值最小。并且最大值最多也只是son[cur] / 2，因为最坏情况（最难分）也就是一条直线，选中间点就可以了。

算法思路：

直观来说，应该是删除那个儿子数最多的那个节点的。因为，没理由再分一些节点给最大的那颗子树把，这样只会更坏。但是却可以把最大的那颗子树分一些节点去另一边，所以优先删除最大的那颗子树的重心，然后判断是否符合要求，不符合就只能暴力往上找了。

判定条件是son[cur] > 2 \* son[重心]就不行。因为这表明son[cur] - son[重心]的值还大于son[cur] / 2。代进去就知道了son[cur] - son[重心] > son[重心]，假设son[cur] = 2 \* son[重心]。

那么son[重心]的最大值是son[cur] / 2，这是不行的。

**回溯处理**：先找当前这个子树中，节点数最大的那个儿子的“重心”，然后暴力判断向上爬就行了。处理到root的时候，后面的已经处理好的了，这就是回溯。

**void** dfs(**int** cur, **int** from) {

son[cur] = 1; //自己算一个节点

ans[cur] = cur; //叶子节点

**int** mx = -inf, pos = cur; //以这个点为子树的儿子数最多的那个pos

**for** (**int** i = first[cur]; i; i = e[i].tonext) {

dfs(e[i].v, cur);

son[cur] += son[e[i].v]; //加上儿子的节点个数

**if** (mx < son[e[i].v]) { //不能算自己，只能算儿子的max

mx = son[e[i].v];

pos = e[i].v; //儿子数最多的那个节点，

}

}

ans[cur] = ans[pos]; //ans[pos]已经算出来了，ans[pos]表明是pos节点的重心

**while** (son[cur] > 2 \* son[ans[cur]]) { // 放缩：son[cur] = 2 \* son[重心]，就不行了

ans[cur] = fa[ans[cur]]; //暴力往上找

}

}

(一)、树中所有点到某个点的距离和中，到重心的距离和是最小的；如果有两个重心，那么他们的距离和一样。

(二)、把两个树通过一条边相连得到一个新的树，那么新的树的重心在连接原来两个树的重心的路径上。

(三)、把一个树添加或删除一个叶子，那么它的重心最多只移动一条边的距离。

树的直径。

从任何一个点出发，bfs走到最远的路，然后从终点再bfs走一次，那就是直径。

**int** tree\_diameter(**int** begin, **bool** flag) {

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

queue<**struct** bfsnode> que;

que.push(bfsnode(begin, 0));

vis[begin] = **true**;

**int** to = begin, mx = 0;

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

**struct** bfsnode t = que.front();

que.pop();

**for** (**int** i = first[t.cur]; i; i = e[i].tonext) {

**int** v = e[i].v;

**if** (vis[v]) **continue**;

vis[v] = **true**;

que.push(bfsnode(v, t.cnt + e[i].w));

**if** (mx < t.cnt + e[i].w) {

to = v;

mx = t.cnt + e[i].w;

}

}

}

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

**else** **return** to;

}

## 4、图的割点、割边、有向图的强连通分量

DFN[i] 其实就是一个标记，表示i号顶点第几个被访问的

low[i] 表示i号顶点在不经过它枚举过来的那个**爸爸**的情况下，能访问到的最远祖先。

割点算法，**注意根结点起码要有两个孩子，才算是一个割点。**算法中，如果cur能访问一个已经被访问的点，而且这个点又不是cur的father，那么只能是cur的祖先。例如1🡪2🡪3🡪4然后4反问了2，不是爸爸3，所以只能是4的祖先了。

**void** dfs (**int** cur, **int** father) {

**int** child = 0;//代表cur有多少个儿子

when++;//这个我们称为时间戳

DFN[cur] = when;//刚开始的辈分

low[cur] = when;//刚开始能访问到的，只能是自己

**for** (**int** i = 1; i <= n; i++ ) { //循环一个图

**if** (e[cur][i] == 1) { //如果cur去i是有路的

**if** (DFN[i] == 0) { //就是i还没被访问过。先跳去看else

child++;//儿子数+1

dfs (i, cur);//分清楚谁是爸爸，谁是儿子，此时很明显cur是i的爸爸

low [cur] = min(low[cur], low[i]);//更新我和儿子能访问到的人

**if** (cur != root && low[i] >= DFN[cur]) { //此时儿子是i 父亲是cur

flag[cur] = 1; //我是割点

}

**if** (cur == root && child == 2) { //如果我是第一辈，首先回溯到有两个儿子

flag[cur] = 1; // 或者变成标记这条边是割边。

}

} **else** **if** (i != father) {

low[cur] = min(low[cur], DFN[i]);//拜师中

}

}

}

}

dfs(1, 1);

割边算法

当我们low[i] > DFN[cur]的时候，表明我们连爸爸都去不到，这个就是割边。

**有向图的强连通分量：**在有向图G中，如果两个顶点间至少存在一条路径，称两个顶点强连通(strongly connected)。如果有向图G的每两个顶点都强连通，称G是一个强连通图。非强连通图有向图的极大强连通子图，称为强连通分量(strongly connected components)。求解这个可以用Tarjan算法。

id[u]：表示顶点u在那一个强连通子图中

sum[id]：表示id这个强连通子图的顶点个数。

**void** tarjan(**int** cur, **int** fa) {

DFN[cur] = low[cur] = ++when; //时间戳

st[++top] = cur; //进栈

vis[cur] = **true**;

**for** (**int** i = first[cur]; i; i = e[i].tonext) {

**int** v = e[i].v;

**if** (!DFN[v]) { //没访问过

tarjan(v, cur);

low[cur] = min(low[cur], low[v]);

} **else** **if** (vis[v]) { // 访问过，而且还在栈里

low[cur] = min(low[cur], DFN[v]);

}

}

**if** (low[cur] == DFN[cur]) { //这个是强连通分量的根节点。

++toSelId;

**do** {

id[st[top]] = toSelId; //块id

sum[toSelId]++; //id节点个数

*// printf("%d ", st[top]);*

vis[st[top]] = **false**;

top--;

} **while** (cur != st[top + 1]);

*// printf("\n");*

}

}

**void** sloveTarjan(**int** n) { //防止开始枚举的节点没有出边，所以要暴力枚举每一个节点

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

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

memset(vis, 0, **sizeof** vis); //标记是否在栈里

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

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

top = when = toSelId = 0;

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

**if** (!DFN[i]) { //每一个没访问过的点都应该tarjan一下

tarjan(i, i);

}

}

}

**无向图的双连通分支：**在图G的所有子图G'中，如果G'是双连通的，则称G'为双连通子图。如果一个双连通子图G'它不是任何一个双连通子图的真子集，则G'为极大双连通子图，双连通分支(biconnected component)，或重连通分支，就是图的极大双连通子图。特殊的，点双连通分支又叫做块。**分为点双连通和边双连通。**

**构造双连通图**

一个有桥的连通图，如何把它通过加边变成边双连通图？方法为首先求出所有的桥，然后删除这些桥边，剩下的每个连通块都是一个双连通子图。把每个双连通子图收缩为一个顶点，再把桥边加回来，最后的这个图一定是一棵树，边连通度为1。

统计出树中度为1的节点的个数，即为叶节点的个数，记为leaf。则至少在树上添加(leaf+1)/2条边，就能使树达到边二连通，所以至少添加的边数就是(leaf+1)/2。具体方法为，首先把两个最近公共祖先最远的两个叶节点之间连接一条边，这样可以把这两个点到祖先的路径上所有点收缩到一起，因为一个形成的环一定是双连通的。然后再找两个最近公共祖先最远的两个叶节点，这样一对一对找完，恰好是(leaf+1)/2次，把所有点收缩到了一起。

## 5、二分图匹配

最小顶点覆盖：用最少的点，让每条边都至少和其中一个点关联

最小边覆盖：用最少的边，让每个顶点都至少和其中一条边关联

最大独立集：在图G中选出m个点，使这m个点两两之间没有边的点中，m的最大值。

任意图的话，求上面的解都是NP难问题。

二分图的话，有下面性质：

①、二分图中最小顶点覆盖 = 最大匹配数，

②、二分图中最小边覆盖 = 顶点数 - 最小顶点覆盖（最大匹配）

③、二分图中最大独立集 + 最小顶点覆盖（最大匹配） = 顶点数

Knm代表左边顶点数是n，右边顶点数是m的二部完全图。就是有n \* m条边。

Kn代表n阶完全图，有n \* (n - 1)条边，如果无向就除以2。

二分图最大匹配：**二分图中边集的数目最大的那个匹配**

匈牙利算法，复杂度O(nm)。n是顶点数，m是边数

有一种很特别的图，就做二分图，那什么是二分图呢？就是能分成两组，S, T。其中，S上的点不能相互连通，只能连去T中的点，同理，T中的点不能相互连通，只能连去S中的点。这样，就叫做二分图。但是他们能否相连是有条件的，不是每个S都能连去T的，所以我们一般要用个isok(S,T)，或者用邻接矩阵e[S][T]==1来判断他们能否相连。现在我们要求的是这个图的最大匹配数量。

FZU 2232 炉石传说

给出自己队伍n<=100个人，每个人两个数据，第一个是生命值，第二个是攻击力。敌方也是n个人，也是一样给出。现在要求是否存在这样的一种情况，自己的人去选人打，要求打到的对方必死，但是自己不能死（对方攻击力不能大于我的）。当攻击力>=生命的时候，就能打死。 (选人打，一个只能选一个). 对于每个敌人，都要选一个自己人来打。这里有点小技巧就是，不需要用邻接矩阵来存图了，直接用个函数来判断能否连通就可以了。

**多case记得memset (match)**

#include <cstdio>

#include <cstdlib>

#include <cstring>

**int** n;

**const** **int** maxn = 1e2 + 20;

**struct** data {

**int** live;

**int** att;

} a[maxn], b[maxn];

**bool** book[maxn];

**int** match[maxn];

**int** isok(**int** i, **int** u) {

**if** (a[i].att >= b[u].live && a[i].live > b[u].att) { //这样才能连通

**return** 1;

}

**return** 0;

}

**int** dfs(**int** u) { *//u是敌人*

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

**if** (book[i] == 0 && isok(i, u)) { //isok(i,u) 代表那里能连一条边

book[i] = 1; //标记后，下一次的dfs不会访问自己

//为什么会进入dfs?因为match[i]有人了，然后叫那个人去找其他人

**if** (match[i] == 0 || dfs(match[i])) {

match[i] = u; //搭配u。第i个自己人，选了打u这个敌人。

**return** 1;

}

}

}

**return** 0;

}

**int** hungary() { //匈牙利算法，多case要memset match

memset (match, 0, **sizeof**(match)); //多case这个记得memset

**int** ans = 0;

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

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

**if** (dfs(i)) ans++;

}

**return** ans;//看看最大能匹配多少个，n个自己，n个敌人。返回(n+n)/2就证明够打了

}

**void** work() {

scanf ("%d", &n);

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

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

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

scanf ("%d%d", &b[i].live, &b[i].att);

**if** (hungary() == n) //匹配数目刚好是n，每个人都匹配了

printf ("Yes**\n**");

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

**return** ;

}

**int** main() {

**int** t;

scanf("%d", &t);

**while** (t--) work ();

**return** 0;

}

二分图最佳匹配：KM算法。带权图的最大匹配。得到最大权值

给定一个n\*n(n<=16)矩阵，e[i][j]代表驾驶员i和导航员j的默契值，要求输出最大默契值

如果不是完美匹配的话，就是两边人的个数不一样，就会TLE。所以我们增加一些点，使得它成为完美匹配，这些增加的点的权值应该为0，表示他们对答案是没有贡献的。就是相当于没匹配了，因为本来就会有一些人是不能匹配的。

★、图中原本不连通的点，要用0来表示，不能用inf来表示，理由也是一样的。

★、如果想输出最小搭配，则大于号那些对应改变，inf也改成-inf即可

例题：玲珑杯ACM 1047 Best couple

**int** match[maxn];//match[col] = row

**int** vx[maxn], vy[maxn];

**int** fx[maxn], fy[maxn];

**int** n, m;

**int** dfs (**int** u) {

vx[u] = 1;

**int** i;

**for** (i = 1; i <= m; i++) { //筛选n个 导航员 col的值

**if** (vy[i] == 0 && fx[u] + fy[i] == e[u][i]) {

vy[i] = 1;

**if** (match[i] == 0 || dfs(match[i])) {

match[i] = u; // match[col]=row;

**return** 1;//搭配成功

}

}

}

**return** 0;//我找不到啊，后面，就会执行km

}

**void** do\_km() { //

**int** i, j;

**int** d = inf; //**改成-inf**

**for** (i = 1; i <= n; i++) { //遍历每一个驾驶员 row的值

**if** (vx[i] == 1) {

**for** (j = 1; j <= m; j++) { //对他进行遍历导航员 col的值

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

**if** (d > (fx[i] + fy[j] - e[i][j])) { //**改成小于号**

d = fx[i] + fy[j] - e[i][j];

}

}

}

}

}

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

**if** (vx[i] == 1) {

fx[i] -= d;

vx[i] = 0; //请0

}

**if** (vy[i] == 1) { //

fy[i] += d;

vy[i] = 0; //情0

}

}

**return** ;

}

**int** anskm() {

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

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

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

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

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

//km算法的一部分，先初始化fx，fy

**for** (**int** i = 1; i <= n; i++) { //遍历每一个驾驶员 row的值

fy[i] = 0;

fx[i] = -inf; //无穷小，**改成inf**

**for** (**int** j = 1; j <= m; j++) { //遍历每一个导航员 col的值

**if** (fx[i] < e[i][j]) { //默契值，**改成大于**

fx[i] = e[i][j];

}

}

}

**for** (**int** i = 1; i <= n; i++) { //遍历每一个驾驶员 row的值

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

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

**while** (!dfs(i)) {//如果他找不到搭配，就实现km算法

do\_km();//km完后，还是会对这个想插入的节点进行dfs的,因为他还没搭配嘛

}

}

**int** ans = 0;

**for** (**int** i = 1; i <= m; i++) //遍历导航员，col的值

ans += e[match[i]][i];//输入的row是驾驶员，col是导航员

//match[i]:导航员i和驾驶员match[i]搭配了 match[col]=row;

**return** ans;

}

总结：上面两个算法，如果你枚举敌人，得到的match[i]=u就是自己人i去打u这个敌人。如果你枚举驾驶员row，得到的就是导航员col去搭配驾驶员row。因为dfs谁，就是为谁找搭配。而这个搭配，用match[i]表示的话。i是它对面的图。没枚举的图。**match[x]=枚举值**

## 6、欧拉回路 && 哈密顿回路

**欧拉图：一定要访问这个图的每条边，而且每条边只能访问一次，能回到起点的图。**

**判断的时候需要先用并查集来判断图是否联通**。也可以用Degree[i] = 0来判断不联通

无向图：

欧拉回路：所有顶点的度数应该都是偶数。（必须是一进一出，所以必定是偶数）

欧拉通路：有且仅有两个顶点的度数是奇数，其他的都是偶数。

有向图：

欧拉回路：每个顶点的入度等于出度。

欧拉通路：起点的入度比出度少一，终点的出度比入度少一。其他点的入度和出度相同。

**哈密顿图：**要求的是经过所有顶点且只能经过一次，能回到起点的图。

求解哈密顿**通路**，需要用上拓扑排序。（用在**有向**、**无环**图中，也就是DAG）

拓扑排序其实就是选课安排的排序，要修C这门课，要求先修A和B这两门课，那么把整个图线性化后（也就是拓扑排序后），先修的课程需要出现在后修的课程的前面。可知A和B没有先修课程，所以这时候拓扑排序的结果不唯一。如果这个图有环，就不存在解。

**bool** DAG\_sort() { //in[u]表示点u的**入度**

**for** (**int** i = 1; i <= n; ++i) { //节点号小的在前

**if** (in[i] == 0) que.push(DAG(i)); //优先队列维护即可

}

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

**int** cur = que.top().cur; //顶点参数

que.pop();

ans.push\_back(cur);

**for** (**int** i = first[cur]; i; i = e[i].tonext) {

**int** v = e[i].v;

in[v]--;

**if** (in[v] == 0) que.push(DAG(v));

}

}

**if** (ans.size() < n) **return** **false**; //有环

**return** **true**;

}

那么如果这个DAG上的任意两个顶点都是全序关系，也就是任意两个顶点都能确定一个关系，也就是单向连通，也就是事件发生的先后关系。那么拓扑排序后的结果是唯一的。比如上面的，A和B是不能确定关系的，也称偏序关系。如果是全序关系，那么他的拓扑排序结果是唯一的。也是对应的哈密顿路径。（注意非DAG图也有哈密顿路径，这里只讨论DAG图的）。

方法就是，先拓扑排序后，然后对于这个序列，任意两个相邻的点都应该存在边，否则就不是哈密顿通路。

## 杂项

1、图的邻接表去重

ACdream 1236

给定一个可能有重边的无向图，要求找出所有割边，注意重复的边不能算割边。

看看时候重复的时候，可以直接在插入的时候，再遍历所有first[u]。有重复的话把id去掉

或者先建完图，O(m)去重，用个used[v] = u标记顶点v有顶点u去过，不用清空，因为u一定是不同的，删除全部边的话，用个pre[v] = j表示上一条边是谁就行了。44ms

**bool** isok (**int** u,**int** v) { //可以在add边的时候询问一下是否重复，重复就不加上去了

**for** (**int** i=first[u]; i; i=e[i].next) { //问一下u顶点所有边能不能去这个点

**if** (e[i].v==v) {

e[i].id=inf;

**return** **false**;

}

}

**return** **true**;

}

或者O（m）去重

**for** (**int** i = 1; i <= n; ++i) { //枚举每一个顶点

**for** (**int** j = first[i]; j; j = e[j].next) {

**if** (book[e[j].v] == i) { //int book[]，被当前号顶点访问过

e[j].id = inf;

e[pre[e[j].v]].id = inf;

}

pre[e[j].v] = j; //这个顶点的上一条边是j这条边。

book[e[j].v] = i; //这个顶点被i号顶点访问过

}

}

这里的pre[]和book[]都要memset，因为多case可能有相同的边。

如果要保留最短的边，只需要比较当前这条和pre[]那条的大小，然后更新pre[]要注意。

# 计算几何

①、给定n条边长，问能否构成n变形。类比三角形的思路，任意n-1边之和大于第n边，这个能在O(n)时间内算出来，预处理所有的和sum，枚举删除的边a[i]即可。

②、**const double PI = acos(-1.0); 定义PI要用这个**

③、对于坐标的hash，（x, y），可以变成x \* max(n, m) + y。大小只有n \* m + y这个级别。

## 1、基本公式

二维坐标的定义

**struct** coor {

**double** x, y;

coor() {}

coor(**double** xx, **double** yy): x(xx), y(yy) {}

**double** **operator** ^ (coor rhs) **const** { //计算叉积（向量积），返回数值即可

**return** x \* rhs.y - y \* rhs.x;

}

coor **operator** - (coor rhs) **const** { //坐标相减，a-b得到向量ba，

//返回一个向量（坐标形式）

**return** coor(x - rhs.x, y - rhs.y);

}

**double** **operator** \* (coor rhs) **const** { //数量积，返回数值即可

**return** x \* rhs.x + y \* rhs.y;

}

**bool** **operator** == (coor rhs) **const** {

**return** same(x, rhs.x) && same(y, rhs.y); //same的定义其实就是和eps比较

}

}; //记得这里有个分号

二维直线的定义

**struct** Line {

coor point1, point2;

Line() {}

Line(coor xx, coor yy) : point1(xx), point2(yy) {}

**bool** **operator** & (Line rhs) **const** { //判断直线和rhs线段是否相交

//自己表示一条直线，然而rhs表示的是线段

//判断rhs线段上两个端点是否在this直线的同一侧即可，用一侧，就不相交

coor ff1 = point2 - point1; //直线的方向向量

**return** ( ((rhs.point1 - point1)^ff1) \* ((rhs.point2 - point1)^ff1) ) <= 0;

//符号不同或者有0，有0代表有点落在直线上，证明相交

}

};

三维坐标的定义

**struct** coor {

**int** x, y, z; //坐标，也可以表示成向量，向量也是一个坐标表示嘛

coor() {}

coor(**int** xx, **int** yy, **int** zz) : x(xx), y(yy), z(zz) {}

**bool** **operator** & (coor a) **const** { //判断两个向量是否共线，共线返回true

//思路：判断叉积，是否各项系数都是0，叉积是0的话，就是共线的了

**return** (y \* a.z - z \* a.y) == 0 && (z \* a.x - x \* a.z) == 0 && (x \* a.y - y \* a.x) == 0;

}

coor **operator** ^ (coor a) **const** {

//得到两个向量的叉积（就是向量积），返回的是一个向量

//如果是二维的话，就是只有y\*a.z - z\*a.y，其他的没有的。

**return** coor(y \* a.z - z \* a.y, z \* a.x - x \* a.z, x \* a.y - y \* a.x);

}

coor **operator** - (coor a) **const** { //如果是c-d的话，得到向量dc，

**return** coor(x - a.x, y - a.y, z - a.z);

}

**int** **operator** \* (coor a) **const** { //得到两个向量的 数量积，返回整数即可

**return** x \* a.x + y \* a.y + z \* a.z;

}

};

1.1、判断线段是否相交，判断点是否在线段上

**bool** OnSegment(coor a, coor b, coor cmp) { //判断点cmp是否在线段ab上

**double** min\_x = min(a.x, b.x), min\_y = min(a.y, b.y);

**double** max\_x = max(a.x, b.x), max\_y = max(a.y, b.y);

**if** (cmp.x >= min\_x && cmp.x <= max\_x && cmp.y >= min\_y && cmp.y <= max\_y)

**return** **true**;

**else** **return** **false**;

}

**bool** SegmentIntersect(coor a, coor b, coor c, coor d) {

**double** d1 = (b - a) ^ (d - a); //direction(a,b,d);以a为起点，计算ab和ab的叉积

**double** d2 = (b - a) ^ (c - a);

**double** d3 = (d - c) ^ (a - c);

**double** d4 = (d - c) ^ (b - c);

**if** (d1 \* d2 < 0 && d3 \* d4 < 0) **return** **true**;

**else** **if** (same(d1, 0) && OnSegment(a, b, d)) **return** **true**; //如果端点在线段上不算相交的

**else** **if** (same(d2, 0) && OnSegment(a, b, c)) **return** **true**; //就是要严格相交的话，就把这四

**else** **if** (same(d3, 0) && OnSegment(c, d, a)) **return** **true**; //行去掉

**else** **if** (same(d4, 0) && OnSegment(c, d, b)) **return** **true**;

**else** **return** **false**;

}

1.2判断直线是否相交、平行、求交点

**struct** data LineIntersect(Line L1, Line L2) { //判断这两条直线是否平行、重合、相交

//随便开个结构体保存就行，double x,y; int flag;flag表明是否平行或重合

**struct** data ans;

ans.flag = 0;//0表示相交

ans.x = L1.point1.x;

ans.y = L1.point1.y;

**if** (((L1.point2 - L1.point1) ^ (L2.point2 - L2.point1)) == 0) { //这样就起码平行

ans.flag = 1; // 1表示平行吧

//然后在直线1找一点连去直线2，如果还是0，说明重合

**if** (((L2.point2 - L1.point1) ^ (L2.point2 - L2.point1)) == 0)

ans.flag = 2; //2表示重合了

**return** ans;

}

**double** t = ((L1.point1 - L2.point1) ^ (L2.point1 - L2.point2)) /

((L1.point1 - L1.point2) ^ (L2.point1 - L2.point2)) ;

ans.x += (L1.point2.x - L1.point1.x) \* t;

ans.y += (L1.point2.y - L1.point1.y) \* t;

**return** ans;

}

1.3 判断能否构成三角形 只需要枚举任意两边相减<第三边即可

**bool** check (**int** a, **int** b, **int** c) {

**if** (abs(a - b) >= c) **return** **false**;

**if** (abs(a - c) >= b) **return** **false**;

**if** (abs(b - c) >= a) **return** **false**;

**return** **true**;

}

1.4 坐标绕坐标旋转角θ后的值，可以配合矩阵快速幂

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：

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

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

1.5 判断点是否在任意多变形上。（顶点必须按顺序输入，顺时针，逆时针等）

思路：求解y=cmp.y与多边形一侧有多少个交点，奇数就在里面，偶数就在外面，**cmp在边上是不行的**。需要增加判断点是否在边上的特判。2代表在边上，1代表在里面，0代表外

**int** PointInPolygon (coor p[], **int** n, coor cmp) {

**int** cnt = 0; //记录单侧有多少个交点，这里的p[]，必须有顺序

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

**int** t = (i + 1) > n ? 1 : (i + 1); //下标为1要这样MOD

coor p1 = p[i], p2 = p[t];

**if** (OnSegment(p1, p2, cmp)) {

coor t1 = p1 - cmp, t2 = p2 - cmp; //同时要叉积等于0，这是在线段上的前提

**if** ((t1 ^ t2) == 0) **return** 2; // 2表明在多边形上，可以适当省略

}

**if** (cmp.y >= max(p1.y, p2.y)) **continue**; //交点在延长线上和在凸顶点上的都不要

**if** (cmp.y < min(p1.y, p2.y)) **continue**; //交点在凹顶点上就要，这里没取等

**if** (same(p1.y, p2.y)) **continue**; //与cmp.y是平行的

**double** x = (cmp.y - p1.y) \* (p1.x - p2.x) / (p1.y - p2.y) + p1.x;

//求交点 p1.y != p2.y不会除0

**if** (x > cmp.x) cnt++; //只统计一侧的交点

}

**return** cnt & 1; //0表明点在多边形外，1表明点在多边形内

}

一些题目：

**POJ 2318 TOYS 利用叉积判断点在线段的那一侧**

题意：给定n(<=5000)条线段，把一个矩阵分成了n+1分了，有m个玩具，放在为位置是(x,y)。现在要问第几个位置上有多少个玩具。

思路：叉积，线段p1p2，记玩具为p0，那么如果(p1p2 ^ p1p0) (**记得不能搞反顺序，不同的**)，如果他们的叉积是小于0，就是在线段的左边，(**注意这里的p1一定是上端点)**。所以，可以用二分找，如果在mid的左边，end=mid-1 否则begin=mid+1。结束的begin，就是第一条在点右边的线段

**POJ 3304 Segments**

题意：给定n（n<=100）条线段，问你是否存在这样的一条直线，使得所有线段投影下去后，至少都有一个交点。

思路：对于投影在所求直线上面的相交阴影，我们可以在那里作一条线，那么这条线就和所有线段都至少有一个交点，所以如果有一条直线和所有线段都有交点的话，那么就一定有解。

怎么确定有没直线和所有线段都相交？怎么枚举这样的直线？思路就是固定两个点，这两个点在所有线段上任意取就可以，然后以这两个点作为直线，去判断其他线段即可。为什么呢？因为如果有直线和所有线段都相交，那么我绝对可以平移到某个极限的端点位置，再旋转到某个极限的端点位置，也不会失去正解。**Bug点就是枚举的两个点是重点的话，这个直线的方向向量是0向量，这样会判断到与所有线段都相交。~~**

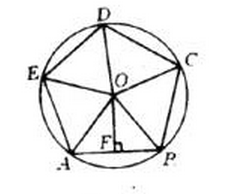
**POJ 1556 The Doors**

题意：给定n堵墙在一个矩形里，现在要你从(0,5)走去(10,5)的最短距离

思路：刚开始还想模拟，就是从(0,5)走，每次x向右一格，然后判断有没和线段相交就可以。但是它的们有可能是小数形式给出的，这样就GG了(x--x+1中可能存在很多门)。正确的方法应该是建图，对于所有门，他们都有端点的，先把他们加入到图中，包括起点的话，一共有num个点吧。然后暴力判断e[i][j]是否能到达就可以，这里用线段相交就可以判断。然后floyd一下就好。bug点：门的端点不应该加进来，就是(x,0)、(x,10)这样的点不应该加入图中，因为那个是死角，不能出去了。

## 2、正N边形公式

①、在正多边形中，只有三种能用来铺满一个平面而中间没有空隙，这就是正三角形、正方形、正六边形。



∠AOP，是中心角，值为ao=2\*PI/n (360/n/360\*2\*PI)

AO是外接圆的半径，也称为正N边形的半径

面积公式：

把正N边形分成N个小三角形，计算出对应的高即可

**double** acreage(**int** n, **double** len) { *//顶点数是n，每条边长是len*

//n>=3

**double** ao = PI / n; //中心角的一半(中心角是PI/n,弧度制)

**double** h = len / 2 / tan(ao);

**return** 1.0 / 2 \* len \* h \* n;

}

内角：

正n边形的内角和度数为：(n-2)×180°;

正n边形的一个内角是(n-2)×180°÷ n.

外角：

正n边形外角和等于n\*180° － (n-2)\*180°=360°

所以正n边形的一个外角为：360°÷ n。

所以正n边形的一个内角也可以用这个公式：180°-360°÷ n.。

中心角：

外接圆，每条边所对的圆周角，360/n。这个角和内角是互补的

对角线：

对角线数目：**n\*(n-3)/2;**

最长对角线长度

**double** diagonal(**int** n, **double** len) {

//求出正N边形的最长对角线长度，每条边长度是len

//目标：求出正N边形外接圆直径，\*\*所有正N边形都有外接圆\*\*

//中心角:360/n..就是每条边对应的圆心角是相等的，有n条边平分

//360/n 转化成弧度制再取一半，ao=PI/n; ans=len/2/sin(ao)\*2

**return** len / sin(PI / n);

}

## 3、平面最近对

分治算法，分开两部分，算出DL表示左边的最近点对，DR是右边，但是可能是从左边选一个点，右边选一个点会更小，这个时候可以暴力。记d=min(DL,DR)。在[L,R]中选出离中间那个分割点a[mid]的x距离小于d的先，大于d的明显不是最优。假设有k个，然后暴力n²枚举每个k，看看能不能更新答案，这个时候明显也是两个点的y距离小于d才能更新，这样一来，每个k，最多不会超过6个点去匹配。

这里两个正方形，边长为d的，不会存在那个点Q，**因为破坏了最短距离**

Q

**是d这样的前提。**那么枚举k的话，只会枚举与他y距离<d的。就是那

K 正方形的6个顶点了。

**struct** coor {

**double** x, y; //这里可以加上一个int id;表明是第几个，防止排序后乱，这样找最近点对

} a[maxn], t[maxn], mi\_a, mi\_b; //t是用来修改的临时的。

**double** mi;//这个用来记录最小值，更新后，每个case需要重新变为inf。不然wa

**void** update(**double** now, **struct** coor a, **struct** coor b) {

**if** (mi > now) {

mi = now;

mi\_a = a;

mi\_b = b;

}

**return** ;

}

**bool** cmpxy(**struct** coor a, **struct** coor b) {

**if** (a.x != b.x) **return** a.x < b.x; //先排好序，不影响位置的，

**else** **return** a.y < b.y; //只会改变它是第几号点

}

**bool** cmpy(**struct** coor a, **struct** coor b) {

**return** a.y < b.y;

}

**double** dis(**struct** coor a, **struct** coor b) { //根据什么规则来选取，可以变换

**return** ((a.x - b.x) \* (a.x - b.x) + (a.y - b.y) \* (a.y - b.y));

}

**double** closepoint(**int** begin, **int** end, **bool** flag) {

**double** d = inf; //选出最小值的

**if** (begin == end) **return** inf; //只有一个点的话，不行

**if** (begin + 1 == end) {

**double** ff = dis(a[begin], a[end]);

**if** (flag) update(ff, a[begin], a[end]); //更新最近对

**return** dis(a[begin], a[end]); //两个点直接算

}

**int** mid = (begin + end) / 2;

**double** DL = closepoint(begin, mid, flag); //选出左右最小p1,p2

**double** DR = closepoint(mid + 1, end, flag); //后面回溯算中间

d = min(DL, DR);

**int** k = 0;

**for** (**int** i = begin; i <= end; i++) { //虽然扫描整个区间，但只有左右些小才有戏

**if** (fabs(a[mid].x - a[i].x) < d) { //选出离中线小于d的。才有戏

t[++k] = a[i];

}

}

sort(t + 1, t + 1 + k, cmpy); //根据y排序下，刚才那个是优先排x的

**for** (**int** i = 1; i <= k - 1; ++i) { //现在暴力枚举这些点的距离

**int** f = 0;

**for** (**int** j = i + 1; j <= k && (t[j].y - t[i].y < d); ++j) { //y相差大于d也没戏

//对于每一个k，满足的点最多也只是6个。上下两个d的正方形，6个点

**if** (d > dis(t[i], t[j])) {

d = dis(t[i], t[j]);

**if** (flag) update(d, t[i], t[j]); //更新最近点对

}

}

}

**return** d;

}

sort(a + 1, a + 1 + n, cmpxy); //先保证按x排好序，不行再按y排序

closepoint(1, n, 0); //0代表不需要找到那两个点

上面那个dis是可以变换的，有一道题目就是平面上有n个点，以每个点为中心作一个正方形，设边长为k，任意两个正方形不得重叠。要你求出最短的k。

注意到任意两个点(x1,y1)、(x2,y2)。他们能作的正方形边长最大是max(abs(x1-x2),abs(y1-y2));

因为可以以最长距离各分一半边长过去即可。现在就是要使得这个值满足所有条件。（要使这个值最大，在上面已经max过了。）直接修改dis函数即可。按照这个规则来找。

**double** dis (**struct** coor a, **struct** coor b) { //根据什么来选取，可以变换

**return** (max(fabs(a.x - b.x), fabs(a.y - b.y)));

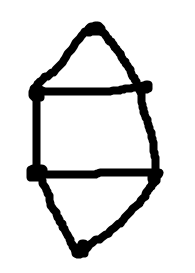
//return sqrt((a.x - b.x) \* (a.x - b.x) + (a.y - b.y) \* (a.y - b.y));

}

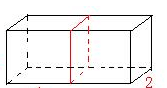
## 4、欧拉公式，分割平面

V：顶点数 E：边数 F：面的数目

在二维的**平面图**中（不能有相交的边），有………………… V – E + F = 2

 V = 6 E = 8 F = 4。 F的时候算上一个外面的无限平面

在三维非闭合空间里，有………………… V – E + F – T = 1 其中T代表三维空间里体的个数

 这里的话，就是 12 – 20 + 11 – 2 = 1 ，T=2，两个正方体

# 数学

## 1、组合数Cnm 防溢出公式

1、如果，(n&m)==m 那么C(n,m)为奇数，否则为偶数

2、求解C(n,0)、C(n,1)……C(n,n)有多少个奇数：ans=1<<(n二进制中1的个数)

3、Cn­­­­­­­­­­­­ – 1m - 1 + Cn – 1m = Cnm

4、求组合数的时候可能会溢出，这个时候我们可以边乘边除，来防止溢出。

因为**Cnm =  = **

那么把上面的1用i来循环，从右到左计算即可

组合数是很大的，C(100,50)也会爆ULL，这个只能求些小的数，例如C(1e8,4)也不会爆。

LL C(LL n, LL m) {

**if** (n < m) **return** 0; *//防止sb地在循环*

**if** (n == m) **return** 1; *//C(0,0)也是1的*

LL ans = 1;

LL mx = max(n - m, m); *//这个也是必要的。能约就约最大*

LL mi = n - mx;

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

ans = ans \* (mx + i) / i;

}

**return** ans;

}

**Lucas 定理** 解决很大的组合数问题 时间O（logp（n）\*p）**用在%很小的数比较有用**。

求解C(n,m)%p 其中：n, m, p (1 <= m <= n <= 10^9, **p是质数且p <= 1e5**)

当MOD的数真的很小，MOD = 110119的话，可以预处理阶乘，**这样快很多。**

LL C(LL n, LL m, LL MOD) {

**if** (n < m) **return** 0; //防止sb地在循环，在lucas的时候

**if** (n == m) **return** 1 % MOD;

LL ans1 = 1;

LL ans2 = 1;

LL mx = max(n - m, m); //这个也是必要的。能约就约最大的那个

LL mi = n - mx;

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

ans1 = ans1 \* (mx + i) %MOD;

ans2 = ans2 \* i % MOD;

}

**return** (ans1 \* quick\_pow(ans2, MOD - 2, MOD) % MOD); //这里放到最后进行,不然会很慢

}

LL Lucas(LL n, LL m, LL MOD) {

LL ans = 1;

**while** (n && m && ans) {

ans = ans \* C(n % MOD, m % MOD, MOD) % MOD;

n /= MOD;

m /= MOD;

}

**return** ans;

}

NEFU 628

求解：C(n,m)%p的值。n, m and p (1 <= n, m, p <= 10^5)。 ★**并且p有可能是合数**

思路：设X = C(n, m) % p，那么X肯定可以分解成p1a \* p2b …. \* pzz这样的东西，那么把最后每个质因子剩下的个数算出来，进行快速幂对p取模即可。这里只进行了乘法，无须判断是否有逆元。 快速幂那里没有进行求逆元操作。

LL calc(LL n, **int** p) {

LL ans = 0;

**while** (n) {

ans += n / p; // 2的倍数贡献一个2，然后4的倍数继续贡献一个。

n /= p;

}

**return** ans;

}

LL solve(LL n, LL m, LL p) {

LL ans = 1;

**for** (**int** i = 1; i <= total && prime[i] <= n ; i++) {

LL t = calc(n, prime[i]); //calc是算出n!中有多少个prime[i]这个因子。

t -= calc(n - m, prime[i]);

t -= calc(m, prime[i]); // t最小也是0

**if** (t) { // 就是当t不是0的时候

ans \*= quick\_pow(prime[i], t, p);

ans %= p;

}

}

**return** ans;

}

有时候，对于p很少的情况p<=1e4，然后我们数据大T<=10000，这样，我们可以预处理出fac[i][j]表示 (j的阶乘)%prime[i]的值。inv[i][j]表示 (j!)关于prime[i]的逆元。然后O(1)处理。注意的是这个公式的话，fac[1][2]以及后面那些fac[1][3]…..都是0的，因为很简单，你如果阶乘中有数字>=prime[i]，那么%prime[i]后结果都是0。但是这样的后果就是C(6,2)%2等于0了。所以这里的组合数要用Lucas辅助来求得。(只能用Lucas，Lucas能避免这个情况)

int fac[maxn][maxn]; // fac[i][j]表示(j!)%prime[i]的值 j<prime[i],如果j==prime[i]，后面的都是0

int inv[maxn][maxn]; // inv[i][j]表示 (j!)对prime[i]求逆元

**void** init() {

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

fac[i][0] = 1;

inv[i][0] = 1; // (0!)=1

**for** (**int** j = 1; j < prime[i]; j++) { //等于prime[i]的话，%后是0了，没用

fac[i][j] = (j \* fac[i][j - 1]) % prime[i];

inv[i][j] = quick\_pow(fac[i][j], prime[i] - 2, prime[i]);

}

}

**return** ;

}

**int** C(**int** n, **int** m, **int** MOD) {

**if** (m > n) **return** 0;

**if** (m == n) **return** 1;

**int** pos = book[MOD]; //book[prime[i]]=i;表明这个素数下标是几多

**return** (fac[pos][n] \* (inv[pos][n - m] \* inv[pos][m] % MOD)) % MOD;

}

这里想得到C(6, 2)%2的话。要调用Lucas(6, 2, 2);

求解C(n, m) % p，其中n, m <= 1e18 && p <= 1e6并且p可能是合数。

扩展Lucas定理 + 中国剩余定理合并。

要求解C(n, m) % p，可以把p拆分成p1a \* p2b ….\* pkz这样的形式，然后分别求解C(n, m) % pix后，得到了模数是pix，余数是C(n, m) % pix的同余方程组。由于每个pix是互质的，所以用CRT求得的最小的整数解就是答案。

比如：C(10, 3) % 14。C(10, 3) = 120, 14有两个质因数2和7，120 % 2 = 0, 120 % 7 = 1,这样用(2, 0)和(7, 1)找到最小的正整数8即是答案，即C(10, 3) % 14 = 8。

但是如果pix中的x >= 2，那模数也不是质数，怎么办？但是这个模数却是质数的x次方，是有方法解的。回顾组合数的阶乘公式，如果能算出某个数n! % pix的值，那么用求逆元的方法就可以求得整个C(n, m) % p的值。**注意这里不是直接求n! % pix的值，是会把pi因子提取出来另外计算，从而使得必定存在逆元。因为必定互质。**

过程是这样的：假设是求19! % 32，虽然这里答案是0，但是实际会把因子3都跳过不算

等价于（1 \* 2 \* 3 \* 4 \* 5 \* 6 \* 7 \* 8 \* 9 \* 10 \* 11 \* 12 \* 13 \* 14 \* 15 \* 16 \* 17 \* 18 \* 19）% 32

等价于（1 \* 2 \* 4 \* 5 \* 7 \* 8 \* 10 \* 11 \* 13 \* 14 \* 16 \* 17 \* 19）\* 36 \* (1 \* 2 \* 3 \* 4 \* 5 \* 6) % 32

然后注意到后面的，是n / pi的阶乘，所以这里可以递归求解。前面的，是有循环节的。每pix为一段，一个循环。比如1 \* 2 \* 4 \* 5 \* 7 \* 8是一个循环，10 \* 11 \* 13 \* 14 \* 16 \* 17也是一个循环。注意到，它们% pix的值是相等的。所以这里也可以快速算出。那么可能最后面不够pix个，剩下的就暴力可以了，复杂度不大。不会超过pix个。最后，就是36怎么解决的问题，也是问题的关键。如果不断递归求解下去，那么最终会是38。做法是把n!中的pi因子全部提取出来另外算，这样也使得n! % pix后，和pix是互质的，逆元存在。最后用CRT合并一下，到此问题完美解决。注意中间过程爆LL，要及时取模。

LL factorialMod(LL n, LL pi, LL cnt) { //求解n! % pi^cnt

**if** (!n) **return** 1;

LL piPow = quick\_pow(pi, cnt, 7e18), temp = 1;

LL y = n / piPow; //分成y段，不要写在上面，piPow变量还没定义出来。

**for** (LL i = 1; i <= piPow; ++i) { //每piPow为一段，然后每段都同余piPow

**if** (i % pi == 0) **continue**; //pi的倍数早已算出

temp = temp \* i % piPow;

}

//1 \* 2 \* 4 \* 5 \* 7 \* 8和10 \* 11 \* 13 \* 14 \* 16 \* 17模9的结果是一样的

LL ans = quick\_pow(temp, y, piPow); //分成了y段然后同余piPow

**for** (LL i = y \* piPow + 1; i <= n; ++i) { //剩下的数字要暴力，例如19!

**if** (i % pi == 0) **continue**; //pi的倍数早已算出

ans = ans \* (i % piPow) % piPow; //取模两次，i会爆LL

}

**return** ans \* factorialMod(n / pi, pi, cnt) % piPow; //递归求解

}

LL exLucas(LL n, LL m, LL p) { //扩展lucas定理

**if** (n <= m) **return** 1 % p;

**int** lenMod = 0;

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

**if** (p % i == 0) { //i是p的质因子

**int** cnt = 0;

**while** (p % i == 0) {

cnt++;

p /= i;

}

++lenMod;

MOD[lenMod] = quick\_pow(i, cnt, 7e18);

r[lenMod] = calc(n, m, i, cnt);

}

}

**if** (p > 1) {

++lenMod;

MOD[lenMod] = p;

r[lenMod] = calc(n, m, p, 1);

}

**return** CRT(r, MOD, lenMod);

}

## 2、各种素数筛法

①、【**1】**即不是质数，也不是合数

②、1e5内有 9592个质数。1e6 内有 78498个质数。

③、在一个大于1的数a和它的2倍之间（即区间(a, 2a]中）必存在至少一个素数。

1、Eratosthen筛法 (埃垃托斯特尼筛法)

思路，用check[]标记不可能是质数的数，那么质数的N倍绝对不是质数。

**const** **int** maxn=1e6+20;

**bool** prime[maxn];//这个用bool就够了，

**bool** check[maxn];

**void** init\_prime() { //有可能需要book[1] = 1, 1的最大质因数的1

**for** (**int** i = 2; i <= maxn - 20; i++) {

**if** (!check[i]) { //说明i是质数

prime[i] = **true**;

**for** (**int** j = i; j <= maxn - 20; j += i) { //筛掉i的倍数，其实可以从2 \* i开始。

check[j] = **true**; //那么j就没可能是质数了

//book[j] = i; //表示j的最大质因数是i，不断更新。后面的质因数更大

**//用这个的时候，需要把2 \* i变成i，否则book[2]不行。（改成i了）**

}

}

}

}

复杂度 : 大概是O(3\*n) maxn=1e6+20时。 执行次数：2853708

题目：给定n（n<=1e6）个数，要求找出所有数字中质因数最大的那个数（num<=1e6）。

思路：注意到第二层循环，每次都约掉了某些合数 = 质数\*k这样的倍数，那么就说明那个合数是这个质数的倍数啦，质数是不能再拆的了，所以他分解质因数就必定有这个质数因子了，所以用个数组book[i]表示数字i的最大质因数是谁，即可。

2、Euler筛法 （欧拉筛法）

上面的方法中，质数是2的时候，排除了6等数字，然后质数是3的时候，又再次排除了6这个数字，不合理。优化使得每个合数只会被它最小的质因数筛去。但是记录质数的时候只能是prime[1]=2,prime[2]=3 …… prime[total]= 这样记录了。因为后面要用。

**const** **int** maxn = 1e6 + 20;

**int** prime[maxn];//这个记得用int，他保存的是质数，可以不用开maxn那么大

**bool** check[maxn];

**int** total;

**void** initprime() {

**for** (**int** i = 2; i <= maxn - 20; i++) {

**if** (!check[i]) { //是质数了

prime[++total] = i; //只能这样记录，因为后面要用

}

**for** (**int** j = 1; j <= total; j++) { //质数或者合数都进行的

**if** (i \* prime[j] > maxn - 20) **break**;

check[i \* prime[j]] = 1;

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

//关键，使得它只被最小的质数筛去。例如i等于6的时候。

//当时的质数只有2,3,5。6和2结合筛去了12，就break了

//18留下等9的时候，9\*2=18筛去

}

}

**return** ;

}

复杂度： 大概是O(1.7n) maxn=1e6+20时 执行次数: 1669920

质因数分解：

给定一个数，写成唯一分解形式，90=2\*3\*3\*5

短除法：枚举每一个质数i，如果当前的n能整除i，就把i的质数因子统统约去，然后剩下的n也是一个质数，是最大的质因子。

**关于剩下的n为什么是质数**：如果一个数n是合数，那么它必定有一个因子在[1,]中，因为假设n=a\*b，取i=min(a,b); 那么，n>=i\*i，所以必有一个因子i在中，如果没有的话，则说明这个数是质数了。所以下面约剩的数也是一个质数，由于约质数因子的时候是从2、3、5从小到大约，所以剩下的是最大的质数因子。

加速：预处理所有质数，然后试着去约的时候，就不需试4、8、16那些不可能的啦。

**int** max\_prime\_factor(**int** n) {

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

**if** (prime[i] > (**int**)sqrt(n)) **break**; //相等还是继续的

**if** (n % prime[i] == 0) {

n /= prime[i];

**while** (n % prime[i] == 0) { //约去所有这些质因子

n /= prime[i];

}

}

}

**return** n; //约剩的就是最大质因子

}

如果相等的不继续，那么49，约不到7，会返回49。

## 3、快速幂、矩阵

求解两个大数相乘，中间结果爆long long的算法，例如a,b<=1e18，求解a\*b%MOD后的值，不要以为相乘没爆unsigned long long 那个范围只有1e19，其实就是10倍long long啦.

这个的速度其实并不快，只是把他拆分相乘，这样就能取模防止溢出了。

LL quick\_mul(LL a, LL b, LL MOD) {

//求解 a\*b%MOD的算法 // 原理：2\*19 = 2\*(1+2+16)

LL base = a % MOD;

b %= MOD; // a\*b%MOD 等价于 (a%MOD \* b%MOD) % MOD;

LL ans = 0; //记住是0 因为中间过程是加

**while** (b) {

**if** (b & 1) {

ans = (ans + base) % MOD;

}

base = (base << 1) % MOD; //notice

b >>= 1 ;

}

**return** ans;

}

例题：HDU 5666

ans = (q - 1) \* (q - 2) / 2 直接上模板即可，注意分开q-1，q-2的奇偶，因为mod后不能再除。

**快速幂取模**

求解a^b%MOD后的值，思路就是把b看成二进制数相加2^19=2^(1+2+16)。为什么这样能达到logn的速度呢？因为如果base=2^4，下一次就直接是2^16次方，跳跃的增幅。

注意和快速乘法取模不同的地方是：

①、快速乘法取模那里的base值，是每次只乘2的，因为2\*19 = 2\*(1+2+16)，此时每次将ans加上base值，我们只需加他的base倍，2\*base倍，16\*base即可。

②、快速幂取模那里的base值，是每次乘以base自己的，因为2^19=2^(1+2+16)，它每次是ans乘base的，要使指数增加，则base=base^k，这样乘的时候才能加上k(例如k=16)。

LL quick\_pow(LL a, LL b, LL MOD) { //求解 a^b%MOD的值

LL base = a % MOD;

LL ans = 1; //相乘，所以这里是1

**while** (b) {

**if** (b & 1) {

ans = (ans \* base) % MOD; //如果这里是很大的数据，就要用quick\_mul

}

base = (base \* base) % MOD; //notice。注意这里,每次的base是自己base倍

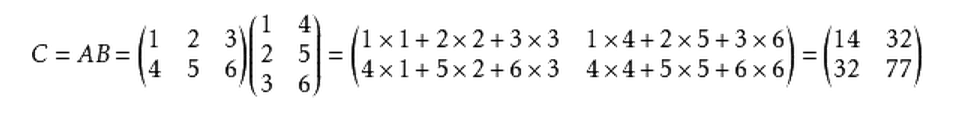
b >>= 1;

}

**return** ans;

}

**矩阵快速幂**



矩阵乘法是满足结合律的，但不满足交换律。就是A\*B\*B\*B 等价于A\*(B)^3。但是，A\*B不等于B\*A；这样可以对B^3进行快速幂取模，从而得到答案。

**struct** Matrix {

LL a[maxn][maxn];

**int** row;

**int** col;

};

//应对稀疏矩阵，更快。

**struct** Matrix matrix\_mul(**struct** Matrix a, **struct** Matrix b, **int** MOD) { //求解矩阵a\*b%MOD

**struct** Matrix c = {0}; //这个要多次用到，栈分配问题，maxn不能开太大，

//LL的时候更加是，空间是maxn\*maxn的，这样时间用得很多，4和5相差300ms

c.row = a.row; //行等于第一个矩阵的行

c.col = b.col; //列等于第二个矩阵的列

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

**for** (**int** k = 1; k <= a.col; ++k) {

**if** (a.a[i][k]) { //应付稀疏矩阵，0就不用枚举下面了

**for** (**int** j = 1; j <= b.col; ++j) {

c.a[i][j] += a.a[i][k] \* b.a[k][j];

c.a[i][j] = (c.a[i][j] + MOD) % MOD; //负数取模

}

}

}

}

**return** c;

}

但是这样话，因为枚举变量的先后次序问题，取模次数又多了，被卡了。下面这个不能应对稀疏矩阵的，但是应对**稠密矩阵**，这个取模次数比较小，所以比较快。快420ms

**struct** Matrix matrix\_mul (**struct** Matrix a, **struct** Matrix b, **int** MOD) { //求解矩阵a\*b%MOD

**struct** Matrix c = {0}; //这个要多次用到，栈分配问题，maxn不能开太大，

//LL的时候更加是，空间是maxn\*maxn的，这样时间用得很多，4和5相差300ms

c.row = a.row; //行等于第一个矩阵的行

c.col = b.col; //列等于第二个矩阵的列

**for** (**int** i = 1; i <= a.row; i++) { //枚举第一个矩阵的行

**for** (**int** j = 1; j <= b.col; j++) { //枚举第二个矩阵的列，其实和上面数值一样

**for** (**int** k = 1; k <= b.row; k++) { //b中的一列中，有“行”个元素 notice

c.a[i][j] += a.a[i][k] \* b.a[k][j]; //**这里不及时取模，又有可能错！**HDU 4565

}

c.a[i][j] = (c.a[i][j] + MOD) % MOD; //如果怕出现了负数取模的话。可以这样做

}

}

**return** c;

}

**struct** Matrix quick\_matrix\_pow(**struct** Matrix ans, **struct** Matrix base, **int** n, **int** MOD) {

//求解a\*b^n%MOD

**while** (n) {

**if** (n & 1) {

ans = matrix\_mul(ans, base, MOD);//传数组不能乱传,不满足交换律

}

n >>= 1;

base = matrix\_mul(base, base, MOD);

}

**return** ans;

}

经典题目：中南大学OJ 1752: 童话故事生成器

X**i** = (a \* X**i-1** + b) % c + 1，其中X1(1<=X1<=1000000) , a(1<=a<=1000000) , b(1<=b<=1000000) , c(1<=c<=1000000) , n(2<=n<=1e18);

对于这样的话，我们可以把那个1放进去，变成X**i** = (a \* X**i-1** + b + 1) % c 这样求出递推矩阵，然后再求出结果的时候，判断下，如果结果是0的话，那么明显是没可能的，结果起码>=1的，这个时候，其实就是(c-1)%c+1=c，因为我们+1了，使得变成了c % c=0，所以这个时候应该输出c。然后，其他的，直接输出即可。C % C + 1 = 1 (C + 1) % C = 1。两者是相等的。

## 4、数字特征、约数个数

**数字特征**：

1：如果数a、b都能被c整除，那么它们的和（a+b）或差(a－b)也能被c整除。

2：几个数相乘，如果其中有一个因数能被某一个数整除，那么它们的积也能被这个数整除。

能被2整除的数，个位上的数能被2整除（偶数都能被2整除），那么这个数能被2整除

能被3整除的数，各个数位上的数字和能被3整除，那么这个数能被3整除

能被4整除的数，个位和十位所组成的两位数能被4整除，那么这个数能被4整除

能被5整除的数，个位上为0或5的数都能被5整除，那么这个数能被5整除

能被6整除的数，各数位上的数字和能被3整除的偶数，如果一个数既能被2整除又能被3整除，那么这个数能被6整除

能被7整除的数，若一个整数的个位数字截去，再从余下的数中，减去个位数的2倍，如果差是7的倍数，则原数能被7整除。例如，判断133是否7的倍数的过程如下：13－3×2＝7，所以133是7的倍数；又例如判断6139是否7的倍数的过程如下：613－9×2＝595 ， 59－5×2＝49，所以6139是7的倍数，余类推。

能被8整除的数，一个整数的末3位若能被8整除，则该数一定能被8整除。

能被9整除的数，各个数位上的数字和能被9整除，那么这个数能被9整除

能被10整除的数，如果一个数既能被2整除又能被5整除，那么这个数能被10整除（即个位数为零）

能被11整除的数，奇数位（从左往右数）上的数字和与偶数位上的数字和之差（大数减小数）能被11整除，则该数就能被11整除。 11的倍数检验法也可用上述检查7的「割尾法」处理！过程唯一不同的是：倍数不是2而是1！

能被12整除的数，若一个整数能被3和4整除，则这个数能被12整除

能被13整除的数，若一个整数的个位数字截去，再从余下的数中，加上个位数的4倍，如果差是13的倍数，则原数能被13整除。如果差太大或心算不易看出是否13的倍数，就需要继续上述「截尾、倍大、相加、验差」的过程，直到能清楚判断为止。

能被17整除的数，若一个整数的个位数字截去，再从余下的数中，减去个位数的5倍，如果差是17的倍数，则原数能被17整除。如果差太大或心算不易看出是否17的倍数，就需要继续上述「截尾、倍大、相减、验差」的过程，直到能清楚判断为止。

另一种方法：若一个整数的末三位与3倍的前面的隔出数的差能被17整除，则这个数能被17整除

能被19整除的数，若一个整数的个位数字截去，再从余下的数中，加上个位数的2倍，如果差是19的倍数，则原数能被19整除。如果差太大或心算不易看出是否19的倍数，就需要继续上述「截尾、倍大、相加、验差」的过程，直到能清楚判断为止。

另一种方法：若一个整数的末三位与7倍的前面的隔出数的差能被19整除，则这个数能被19整除

能被23整除的数，若一个整数的末四位与前面5倍的隔出数的差能被23(或29)整除，则这个数能被23整除

能被25整除的数，十位和个位所组成的两位数能被25整除。

能被125整除的数，百位、十位和个位所组成的三位数能被125整除。

相邻的两个数的平方差一定是奇数。X2 – (x - 1)2 = 2 \* x + 1

相隔的两个数的平方差一定是偶数。(x + 1)2 – (x - 1)2 = 4 \* x

可以应用在给定一个直角三角形的一条边，要求确定直角三角形。

**约数个数**：（思考下枚举约数个数的时候，复杂度会是多少）**直接for，continue不是约数的**

[1, 1e5]。 那个数字是：7560 约数个数是：64

[1, 1e6]。 那个数字是：720720 约数个数是：240

[1, 1e9]。 那个数字是：735134400 约数个数是：1344

[1, 1e18]。那个数字是：897612484786617600 约数个数是：103680

**完美数：**

因子数加起来是自己本身的数，例如6 = 1 + 2 + 3。计算完美数的公式：如果2n - 1是一个质数，那么，由公式N(n)=2(n - 1) \* (2n - 1)算出的数一定是一个完美数。目前还没发现奇完美数。这里不需要n是质数的，需要2^n - 1是一个质数。

## 5、扩展欧几里德算法和求逆元

乘法逆元：满足b\*k≡1 (mod MOD)的k值就是b关于MOD的乘法逆元，k的值可能是b^n。

就是如果我们 求得这样的k，那么，在运算(a/b)%MOD的时候（**前提保证a能整除b，不然会截断的话就没意义**），写成(a\*b^(-1)\*1)%MOD,等价于( [a\*b^(-1)%MOD]\*[1%MOD] )%MOD,然后把那个1%MOD，用b\*k%MOD整体带入，就可以约去b。

现在主要的问题就是知道b和MOD，怎么去求解逆元k的问题了。(扩展欧几里德算法)

首先扩展欧几里德主要是用来与求解线性方程相关的问题，所以我们从一个线性方程开始分析。现在假设这个线性方程为a\*x+b\*y=m，如果这个线性方程有解，那么一定有gcd(a,b) | m，即a，b的最大公约数能够整除m(m%gcd(a,b)==0)。证明很简单，由于a%gcd(a,b)==b%gcd(a,b)==0，所以a\*x+b\*y肯定能够整除gcd(a,b)，如果线性方程成立，那么就可以用m代替a\*x+b\*y，从而得到上面的结论，**利用上面的结论就可以用来判断一个线性方程是否有解。任何时候：a\*x+by=gcd(a,b)都是有解的。并且这个算法解出的|x|+|y|是最小的。**那么，如果我们能求得ax+by=1中的(x,y)，所求得的x就是a关于b的逆元，y就是b关于a的逆元，为什么？两边同时%a或者%b试试?

扩展欧几里德算法：

a\*x + b\*y = gcd(a, b) = gcd(b, a % b) = b \* x + (a % b) \* y

那么把后面的拆分开来，a\*x1 + b\*y1 = b \* x2 + (a - [a / b] \* b)y2;

根据同等 ---> x1 = y2; y1 = x2 - [a / b]y2;

注意前提是a,mod互质，mod相当于ax+by=1的b

**int** exgcd(**int** a, **int** mod, **int** &x, **int** &y) {

//求解a关于mod的逆元 ★：当且仅当a和mod互质才有用

**if** (mod == 0) {

x = 1;

y = 0;

**return** a;//保留基本功能，返回最大公约数

}

**int** g = exgcd(mod, a % mod, x, y);

**int** t = x; //这里根据mod==0 return回来后，

x = y; //x,y是最新的值x2,y2,改变一下，这样赋值就是为了x1=y2

y = t - (a / mod) \* y; // y1=x2(变成了t)-[a/mod]y2;

**return** g; //保留基本功能，返回最大公约数

}

**int** get\_inv(**int** a, **int** MOD) { //求逆元。记得要a和MOD互质才有逆元的

**int** x, y; //求a关于MOD的逆元，就是得到的k值是a\*k%MOD==1

**int** GCD = exgcd(a, MOD, x, y);

**if** (GCD == 1) //互质才有逆元可说

**return** (x % MOD + MOD) % MOD; //防止是负数

**else** **return** -1;//不存在

}

求ans1 / ans2 % MOD的解。

这个得到的逆元比较小，不需要用快速幂。所以ans=(ans1\*get\_inv(ans2,MOD)%MOD); 这里得到的get\_inv那个返回值，就是有ans2\*get\_inv%MOD=1的功能。然后整体代入即可。

**费马小定理**，求b%Mod的逆元k还有另外一种方法，即k=b^(Mod-2)%Mod，因为b^(Mod-1)%Mod=1(**这里需要Mod为素数**)。这样的话b\*k%MOD就等于1了。这个在用的时候  这样：ans = ans1\*quick\_pow(ans2,MOD-2,MOD)%MOD，这是因为它的逆元k=b^(Mod-2)%Mod比较大的缘故，要用快速幂取模。

★、CRT（中国剩余定理）

**条件：要求mod[]任意两个元素要互质。**

定理：ans在% lcm(mod[1]…mod[n])下的解是唯一的。

X % 3 = 2 X % 5 = 3 X % 7 = 2 minans = 23

怎么做呢？可以先把除以3余数是2的数字先写出来：2 5 8 ....再把除以5余数是3的写出来 3、8、....那么共同的数字是8,所以8就是除以3余2且除以5余3的最小数字。。

中国剩余定理也是类似的思想

对于每条等式，都找出一个val，使得val % mod[i] = r[i]且 val % 其他数字是等于0的

第一条式子，这个val = 140

关于怎么找每条式子的val，可以想到用逆元，先解出t % mod[i] = 1，再把r[i] \* t即是val。因为这个时候余数就是r[i]了，（r[i]是余数，肯定小于mod[i]）

r[i] \* t % mod[i] = ((r[i] % mod[i]) \* (t % mod[i])) % mod[i] = r[i]

下面来讨论下为什么要val % 其他数字是等于0的。这里是根据余数的性质，上面的式子，val分别是140、63、30。把他们加起来 % (mod[1] \* mod[2] \* ..... \*mod[n])是答案。先求解前两个的答案，就是先满足除以3余2，除以5余3的数。ans = (140 + 63) % 15。这里的140满足%3 = 2而且%其他数字 = 0。这样的好处是不会相互影响。满足一个等式，而不会影响另一个等式

LL CRT(LL r[], LL mod[], **int** n) { // X % mod[i] = r[i]

LL M = 1;

LL ans = 0;

**for** (**int** i = 1; i <= n; ++i) M \*= mod[i];

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

LL MI = M / mod[i]; //排除这个数

ans += r[i] \* (MI \* get\_inv(MI, mod[i])); //使得MI \* get\_inv(MI, mod[i]) % mod[i] = 1

ans %= M;

}

**if** (ans < 0) ans += M;

**return** ans;

}

当MOD[]不是两两互质时：可以用扩展欧几里德算法合并。

假如那个数字是x，那么有：

x = k1 \* MOD[1] + r[1]

x = k2 \* MOD[2] + r[2]

合并起来，就是：k1 \* MOD[1] – k2 \* MOD[2] = r[2] – r[1]

可以用扩展欧几里德算法求出k1，那么这时候。x = k1 \* MOD[1] + r[1]，同时拥有满足上面两条等式的作用。然后把上面两条等式合并成一条，再和第三条求解。怎么合并成一条？引用定理，ans在 % lcm的情况下解是唯一的，那么我们只要更新模数为LCM(mod[1], mod[2])，余数就是 x % LCM(MOD[1], MOD[2])，再和第三条合并即可。

然后最后的余数就是答案。

**void** work() {

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

cin >> MOD[i] >> r[i];

}

LL ans, tMod = MOD[1], tR = r[1];

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

LL x, y;

**if** (!get\_min\_number(tMod, -MOD[i], r[i] - tR, x, y)) {

cout << "-1" << endl;

**return**;

}

// cout << x << " " << y << endl; 当get\_min\_number取正数

ans = x \* tMod + tR;

tMod = LCM(tMod, MOD[i]); //更新mod数

tR = ans % tMod; //更新余数

}

cout << tR << endl; // % lcm 下的解是唯一的，并且最小

}

不能以为ans是最小的答案。当MOD[] = {4, 3}，r[] = {3, 1}的时候，如果在get\_min\_number里取了正数，就是对于这条方程：4k1 – 2k2 = -2解出的解是（1，3），这个时候ans就是7，需要% lcm(4)，才是最小的答案。

★、**求解方程ax+by=c的最小（正）整数解 (a或者b是负数的话，直接放进去也没问题)**

首先就是如果要求解ax+by=c的话，用exgcd可以求到ax+by=1的x1，y1。那么我们首先把a和b约成互质的（除以gcd(a,b)即可），求到Ax+By=1的x1和y1后，但是我们要的是Ax+By=C的解，所以同时乘上C，得到了ACx + BCy = C的解，可以把C放进x里面，也就是Ax+By = C的解，这里的大写的字母是代表约去gcd(a,b)后的方程。然后这个方程的解就是x0 = x1 + (b / abgcd) \* k。y0 = y1 - (a / abgcd) \* k。k是{-1,-2,0,1,2}等。

**bool** getMinNumber (LL a, LL b, LL c, LL &x, LL &y) { //得到a\*x+b\*y=c的最小整数解

LL abGCD = \_\_gcd(a, b);

**if** (c % abGCD != 0) **return** **false**; //不能整除GCD的话，此方程无解

a /= abGCD;

b /= abGCD;

c /= abGCD;

LL tx, ty;

exgcd(a, b, tx, ty); //先得到a\*x+b\*y=1的解，注意这个时候gcd(a,b)=1

x = tx \* c;

y = ty \* c; //同时乘上c，c是约简了的。得到了一组a\*x + b\*y = c的解。

LL haveSignB = b;

**if** (b < 0) b = -b; //避免mod负数啊，模负数没问题，模了负数后+负数就GG

x = (x % b + b) % b; //最小解

**if** (x == 0) x = b; //避免x = 0不是"正"整数，没要求取正数就不要用这个，**可能会溢出**

y = (c - a \* x); // haveSignB;

**return** **true**; //true代表可以

}

POJ 1061青蛙的约会。求解(x+mT)%L=(y+nT)%L的最小步数T。

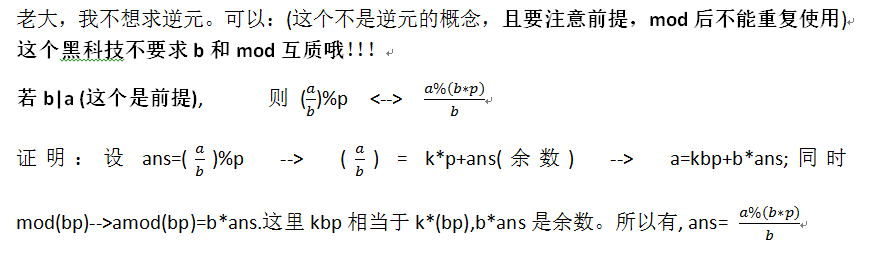
因为是同余，所以就是(x+mT)%L-(y+nT)%L=0。可以写成(x-y+(m-n)T)%L=0。就是这个数是L的倍数啦。那么我可以这样x-y+(m-n)T + Ls = 0。就可以了，s可正可负，就能满足条件。

题目：CodeChef GIVCANDY

题意：甲a个苹果，乙有b个雪梨。现在能给甲c个苹果，或者给乙d个雪梨，使得他们的差值的绝对值尽量小。

题解：设给甲x个苹果，乙y个雪梨。分析后得到方程就是要使得**x\*c-y\*d=(b-a)+K** 有解，并且要使得k最小。明显abs(b-a)%GCD(c,d)==0的话，k是0，否则(b-a)有两种选择。要么减一个数使得它能整除GCD，要么增加一个数。取最小值即可。就是10 MOD 6，可以减去4，也可以增加2。(1 ≤ A, B, C, D ≤ 10^14)

LL dis=abs(a-b); LL ans1=dis%GCD; LL ans2=GCD-dis%GCD; printf ("%lld\n",min(ans1,ans2));



## 6、各种数列（卡特兰数、 递推式组合数）

1、卡特兰数

从第0项开始，依为：1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862。

递推公式：h(n)= h(0)\*h(n-1)+h(1)\*h(n-2) + ... + h(n-1)h(0) (n>=2)

组合数公式： h(n) = //如果MOD的数不是质数，就用递推公式。O(n²)

组合数公式： h(n)= C(2n,n)-C(2n,n+1)

这样就可以解决MOD的数不是质数的问题了。用约质因子，然后快速幂。那个方法。值得注意的是，因为你前后都MOD了，所以可能造成前面的比后面的小，相减会小于0，所以要+MOD来修正一下。( solve(2\*n,n,MOD) - solve(2\*n,n+1,MOD) + MOD ) % MOD

2、递推式，组合数求解 （开始的下标是1，如果不是从1开始，则直接减去偏移量）

题目：玲珑杯1049

1 1 最后一项是直接卡特兰数

1 2 2 然后其他的就是C(n + m – 1, m - 1) – C(n + m – 1, m - 2)

1 3 5 5 注意特判m大于等于2

1 4 9 14 14

题目：FZUOJ 2238 Daxia & Wzc's problem （d的开始位置不同，然后偏移）

a a + d a + 2d a + 3d a + 4d

a 2a + d 3a + 3d 4a + 6d 5a + 10d

a 3a + d 6a + 4d 10a + 10d 15a + 20d

a 4a + d 10a + 5d 20a + 15d 35a + 35d

计算a的系数的公式是：C(n + m – 2, n - 1) 计算d的系数的公式是：C(n + m – 2, m - 2)

因为d是从第2列才开始出现，从第1行就有出现了，那么按照计算a的公式，就是把n和m都偏移去0那里再计算，那么，要把d偏移到第1列开始，所以要m – 2。

## 7、米勒测试和大数分解

米勒测试：根据费马小定理，有一个质数p，那么所有的(1<=a<=p-1)，都有a^(p-1)1(mod p)

成立。那么根据逆否命题，我们列举出一个这样的数a，使得a^(p-1)%p不等于1的话，即可说明p不是质数。Miller-rabin算法就是多次用不同的a来尝试p是否为素数。它是根据一个定理来进行的：**如果对于一个素数p(p已经假定是素数)**，那么(1<=X<=p-1)中，若X^2%p=1,那么X只能是1或者是p-1。所以如果此时的X不是1或者p-1的话，则说明p是合数。那个X不断平方，平方到指数是(p-1)即可停止，所以思路就是，先把需要判定的那个数n，把(n-1)拆分成2^t \* k这种形式。然后随机生成一个数a=rand()%(n-1)+1; 写成a^k次方，把底数看成的X，平方t次，即可到达(n-1)。米勒测试check\_time次，**失败率就是1/(2^check\_time)**

/\*Miller-rabin算法，判断大数字是否为素数\*/

**const** **int** check\_time = 20;

**bool** check(LL a, LL n, LL x, LL t) { //以a为基。n-1写成了 2^t \* k,判断n是否为合数

LL ret = quick\_pow (a, x, n); //先算 a^k%n 后来一直平方.平方t次

LL last = ret; //last就是a^k次方这个值，看成整体。符合X^2这个公式

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

ret = quick\_mul(ret, ret, n); //平方他,last就是一个大X,ret是X^2

**if** (ret == 1 && last != 1 && last != n - 1) **return** **true**; //合数

last = ret;

}

**if** (ret != 1) **return** **true**; //费马小定理，如果a^(n-1)%n != 1就绝逼不是素数

**return** **false**;

}

**bool** Miller\_Rabin(LL n) { //判断n是否质数

**if** (n < 2) **return** **false**;

**if** (n == 2) **return** **true**;

**if** (n % 2 == 0) **return** **false**; //偶数不是质数

LL k = n - 1;

LL t = 0; //把n-1拆成 2^t \* k 这种形式,那么从k开始验证，a^k,不断平方即可

**while** ( (k & 1) == 0 ) { //如果x还是偶数的话，就是还有2的因子

k >>= 1;

t++;

}

**for** (**int** i = 1; i <= check\_time; i++) {

LL a = rand() % (n - 1) + 1; //最大去到n-1,[1,n-1]

**if** (check (a, n, k, t)) //n-1写成了 2^t \* k.米勒测试

**return** **false**; //合数

}

**return** **true**; //质数

}

**pollard\_rho 算法进行质因数分解**。大数质因数分解 复杂度sqrt(p)。p为n约数的个数

用的时候要带上米勒测试一起用（快速判断是否素数）。然后fac[]是100 = 2\*2\*5\*5这样的。

LL factor[500];//质因数分解结果（刚返回时是无序的）

**int** tol;//质因数的个数。数组下标从1开始

LL gcd(LL a, LL b) {

**if**(a == 0)**return** 1;

**if**(a < 0) **return** gcd(-a, b);

**while**(b) {

LL t = a % b;

a = b;

b = t;

}

**return** a;

}

LL Pollard\_rho(LL x, LL c) { //

LL i = 1, k = 2;

LL x0 = rand() % x;

LL y = x0;

**while**(1) {

i++;

x0 = (quick\_mul(x0, x0, x) + c) % x;

LL d = gcd(y - x0, x);

**if**(d != 1 && d != x) **return** d;

**if**(y == x0) **return** x;

**if**(i == k) {

y = x0;

k += k;

}

}

}

//对n进行素因子分解

**void** findfac(LL n) {

**if**(Miller\_Rabin(n)) { //素数

factor[++tol] = n;

**return**;

}

LL p = n;

**while**(p >= n) p = Pollard\_rho(p, rand() % (n - 1) + 1);

findfac(p);

findfac(n / p);

**return** ;

}

## 8、欧拉函数eular

phi[n]表示[1, n]中与n互质的个数。

公式：phi[x] = x \* (1 - 1/p[1]) \* (1 - 1/p[2]) …..(1 - 1/p[n]) 其中p[i]是x的质因子，x是不为0的整数。φ(1)=1（唯一和1互质的数(小于等于1)就是1本身）。 (注意：每种质因数只一个。比如12=2\*2\*3那么φ（12）=12\*（1 - 1/2）\*(1 - 1/3)=4

**1、**欧拉函数是积性函数——若m,n互质，phi(n\*m) = phi(n) \* phi(m);

**2、**特殊性质：当n为奇数时，phi(2 \* n) = phi(n);

**3、当n > 1时，1…n中与n互质的整数和为 n \* phi(n) / 2**

**4、对于一个数n，在(1,n)中与它gcd等于k的个数，就是： eular(n / k) 个; 前提是: k|n**

**证明:**考虑当n约去最大公约数k后，与它互质的数再乘以k，得到的数与n的gcd只能是k

**5、∑d|n phi(d) = n**

**int** eular(**int** n) {

**int** ans = 1, q = (**int**)sqrt(n + 0.5); //ans从1开始

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

**if** (n % i == 0) {

n /= i; //约去了一个，约本来x上的，有剩的while补上，因为eular只算一次

ans \*= (i - 1); //这个是本来分子的

**while** (n % i == 0) {

n /= i;

ans \*= i; //补上这些遗漏的质因子

}

q = (**int**)sqrt(n + 0.5);

}

}

**if** (n > 1) ans \*= n - 1; // 不用除了，这个质因子只有一个，可以在x中约去了

**return** ans;

}

用nlognlogn的算法预处理所有phi[1—n]的值

**void** init\_phi() {

phi[1] = 1;

**for** (**int** i = 2; i <= maxn - 20; i++) {

**if** (!phi[i]) {

**for** (**int** j = i; j <= maxn - 20; j += i) {

**if** (!phi[j]) phi[j] = j;

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

}

}

}

**return** ;

}

求解元素X在区间[1, up]中，有多少个数和X互质。（容斥）

思路：把X质因数分解，多了的不要。12 = 2 \* 3。然后有个明显的道理就是如果是2的倍数的话，那么就一定不会与12互质，所以需要减去2的倍数，减去3的倍数，再加上6的倍数。容斥的思路好想，但是不怎么好写。**所以结果是总数量up – 不互质的个数**。

预处理；his[val][]表示元素val拥有的质因子，Size[val]表示有多少个。记得1是不做任何处理的。就是Size[1] = 0。**Dfs的cur表示下一次从哪里开始，不往回枚举，就是任意k个值。**

**int** calc(**int** up, **int** cur, **int** number, **int** tobuild, **int** flag) { //一开始flag是0。0表示加，1减

**int** ans = 0;

**for** (**int** i = cur; i <= Size[number]; ++i) {

**if** (flag == 0) {

ans += up / (his[number][i] \* tobuild);

} **else** ans -= up / (his[number][i] \* tobuild);

ans += calc(up, i + 1, number, his[number][i] \* tobuild, !flag);

}

**return** ans;

}

计算12在[1, 24]就是**24** - calc(24, 1, 12, 1, 0)。tobuild是选择k个质因数后生成的数字。

复杂度分析：Cn1 + Cn2 + …. + Cnn = 2n – Cn0。所以最多是2 \* 3 \* 5 \* 7 \* 11 \* 13 = 30030

此时复杂度只是：64次。

## 9、AntiPrime

AntiPrime(反素数)，求出区间内约数最多且最小的那个数

首先，数据范围是n<=1e9,数据太大，如何快速算出来呢？我们注意到，如果是暴力算的话，最快的方法就是分解质因子，然后组合式计算啦。但是在算18和30的约数的时候，他们的

gcd(18,30)=6，其实是被重复算了的，那么我们思维反过来一下，把分解质因数变成用质因数去组合，使得变成区间内的数，这样一来，我们在2\*3的时候，\*3就得到了18，\*5就得到了30，能省掉一定的时间。但是还是会TLE。假如我们现在枚举到的数是now，会不会它的约数根本就没可能存在于区间里呢？也就是[begin,end]根本就没这些约数。[7,11]内是不会存在6的倍数的。如果[1,begin-1]中6的约数和[1,end]中6的约数相同，说明什么？新加进去的区间[begin,end]根本就没6的约数，这里可以剪枝。还是TLE！！可行性剪枝，如果一个数是now，现在枚举一个新的质数去乘以它，去结合成新的数字，那么如果它无论组成什么其他数字，因子个数都不会超过当前最优值mx呢？怎么判断呢？放缩咯，假如现在是2\*3，重新去匹配一个新的素数5，那么，我就要看，当前2\*3还能再乘多少个3呢？我记作q，那么这个新的匹配，最理想的情况下因子个数会多2­­q­倍，为什么呢？把那些3，全部替换成5\*7\*11\*13这样来算的话，就是有2q个了。别以为这样没用，当你搜[1,1e9]的时候，你枚举到8000w，再去枚举5那些是没用的，根本就不可能，这里能剪很多。

其实我们还有一个根本的问题没解决，那就是预处理素数到多大，还有万一它是大素数呢？

想着预处理多少，要看数据，预处理出来的最大质数，primeMax‑2是要大过1e9才行的。为什么呢？因为你只有这样，才能防止它数据是两个大质数相乘的形式[primeMax2,primeMax2]。这里的因子个数是3，你枚举不到这个primeMax的话，就只能得到2。

还有那个大素数，没什么怕的，如果当前那个数now，幻想它乘以一个大质数，还是在end的范围的话，就看看\*2和mx谁大咯。乘以一个大素数也才加一倍因子数。其实乘以一个小的质因子的话，因子数会更多，这里主要是判断只有一个大素数的特殊情况。枚举不到那个大素数那里的。

NOJ 1203 //多case的话，记得把mx设置回0，pr = 0;

//cur:当前枚举质数的下标,不用返回来枚举了。

//cnt:分解质因式时：拥有(当前下标那个）素数多少个

//now:当前枚举的那个数字，就是所有质因子相乘得到的数子

//from: 假如:2\*2\*3\*5\*7,然后枚举3，记录的是2\*2，枚举5，记录的是2\*2\*3,

//如果是枚举相同的数，则不用变，因为它记录的是上一个不同的质因子一共拥有的因子数。

//所以乘上(cnt+1)，就是包括上现在这个质因子一共拥有的因子数了。

**void** dfs(**int** cur, **int** cnt, LL now, LL from) {

LL t = from \* (cnt + 1); //现在一共拥有的因子数

**if** (now >= BEGIN && t > mx || t == mx && pr > now && now >= BEGIN) { //有得换了

mx = t;

pr = now;

}

**for** (**int** i = cur; i <= total; ++i) { //枚举每一个素数

LL temp = now \* prime[i];

**if** (END / now < prime[i]) **return** ; //这个数超出范围了

**if** (i == cur) { //没有变，一直都是用这个数.2^k

dfs(cur, cnt + 1, temp, from);

//唯一就是from没变，一直都是用着2，不是新质数

} **else** { //枚举新质数了。

LL k = (cnt + 1) \* from; //现在有K个因子

LL q = (LL)(log(END / now) / log(prime[cur])); //2\*3插入5时，用的是3来放缩

LL add = k \* mypow(2, q);

**if** (add < mx) **return** ; //这里等于mx不return，可以输出minpr

**if** ((BEGIN - 1) / now == END / now) **return**; //不存在now的倍数

**if** (END / now > prime[total]) { //试着给他乘上一个大素数 [999991,999991]

**if** ( k \* 2 > mx ) { //乘以一个大素数，因子数\*2

pr = END;//如果只有一个大素数[1e9+7,le9+7]那么，就是端点值

//否则，是2\*3\*5\*bigprime的话，结果不是最优的，

mx = k \* 2;

}

}

dfs(i, 1, temp, k);

}

}

**return**;

}

scanf ("%lld%lld", &begin, &end); //然后这个now=1，拥有的约数个数就是1个

dfs(1, 0, 1, 1); //刚开始的时候，下标从1开始，拥有这个素数0个，当前数字最少也是1吧

★、出现pr和mx都是1的情况就是你没initprime();

★、求解[1, n]中最小的反素数的高效方法，较小质因子个数永远是较多的

**void** dfs(LL curnum, **int** cnt, **int** depth, **int** up) {

**if** (depth > total) **return** ; // 越界了，用不到那么多素数

**if** (cnt > mx || cnt == mx && pr > curnum) {

pr = curnum;

mx = cnt;

}

**for** (**int** i = 1; i <= up; ++i) { //枚举有多少个prime[depth]

**if** (END / curnum < prime[depth]) **return** ;

**if** ((BEGIN - 1) / curnum == END / curnum) **return** ; //区间不存在这个数的倍数

curnum \*= prime[depth]; //一路连乘上去

dfs(curnum, cnt \* (i + 1), depth + 1, i); // 2^2 \* 3， 3最多2个

}

}

dfs(1, 1, 1, 64); //64表示最大是2^64次方，一般只用前16个素数就够了。

## 10、万能积分公式---simpson

求解任何积分问题，都可以用simpson公式来拟合这段区间內积分的值，只要你设定精度，即可判断是否拟合成功，否则递归分小区间继续拟合。精度太高，则很慢，精度太小，则wa。例题：求半径为r和R的两个圆柱相交的体积。

**double** f(**double** x) {

**double** a = sqrt(r \* r - (r - x) \* (r - x));

**double** ao = asin(a / R);

**return** 2 \* a \* sqrt(R \* R - a \* a) + 2 \* ao \* R \* R;

}

**double** simpson(**double** begin, **double** end) {

**return** (f(begin) + f(end) + 4 \* f((begin + end) / 2)) \* (end - begin) / 6;

}

**double** calc(**double** begin, **double** end) {

**double** mid = (begin + end) / 2;

**double** ans = simpson(begin, end); //用simpson积分拟合这个值

**double** cmp = simpson(begin, mid) + simpson(mid, end);

**if** (fabs(ans - cmp) < eps) **return** ans; // 拟合成功

**return** calc(begin, mid) + calc(mid, end);

}

**void** work() {

**if** (r > R) swap(r, R);

printf ("%0.4lf**\n**", calc(0, r) \* 2);

**return** ;

}

## 11、高斯消元

求解线性方程组，整一个过程，其实就是把**系数矩阵**化成**单位矩阵**，然后对应的增广矩阵那一列，就是对应着整个方程的解。

①、如果是唯一的解，那么增广矩阵那一列就是

②、如果无解，则肯定是存在矛盾式，也就是0 \* X1 + 0 \* X2 + …. + 0 \* Xn = val（val != 0）

③、如果有自由变量，则慢慢讨论。Guass返回的就是自由变量个个数。

★、如果题目要求%8.2lf输出，则不需要添加额外的空格。

**class** GaussMatrix { //复杂度O(n3)

**public**:

**double** a[maxn][maxn];

**int** equ, val; //方程（行）个数，和变量（列）个数，其中第val个是b值，不能取

**void** swapRow(**int** rowOne, **int** rowTwo) {

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

swap(a[rowOne][i], a[rowTwo][i]);

}

}

**void** swapCol(**int** colOne, **int** colTwo) {

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

swap(a[i][colOne], a[i][colTwo]);

}

}

**bool** same(**double** x, **double** y) {

**return** fabs(x - y) < eps;

}

**int** guass() {

**int** k, col; // col，当前要处理的列， k当前处理的行

**for** (k = 1, col = 1; k <= equ && col < val; ++k, ++col) { //col不能取到第val个

**int** maxRow = k; //选出列最大值所在的行，这样使得误差最小。（没懂）

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

**if** (fabs(a[i][col]) > fabs(a[maxRow][col])) {

maxRow = i;

}

}

**if** (same(a[maxRow][col], 0)) { //如果在第k行以后，整一列都是0

--k; //则这个变量就是一个自由变量。

**continue**;

}

**if** (maxRow != k) swapRow(k, maxRow); // k是当前的最大行了

**for** (**int** i = col + 1; i <= val; ++i) { //整一列约去系数

a[k][i] /= a[k][col];

}

a[k][col] = 1.0; //第一个就要变成1了，然后它下面和上面的变成0

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

**if** (i == k) **continue**; //当前这行，不操作

**for** (**int** j = col + 1; j <= val; ++j) { //要使a[i][col] = 0,则需要a[i][col]倍

a[i][j] -= a[i][col] \* a[k][j]; //这一行减去相应的倍数

}

a[i][col] = 0.0;

}

*// debug();*

}

**for** (**int** res = k; res <= equ; ++res) {

**if** (!same(a[res][val], 0)) **return** -1; //方程无解

}

**return** val - k; //自由变量个数

}

**void** debug() {

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

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

printf("%6.2lf ", a[i][j]);

}

printf("**\n**");

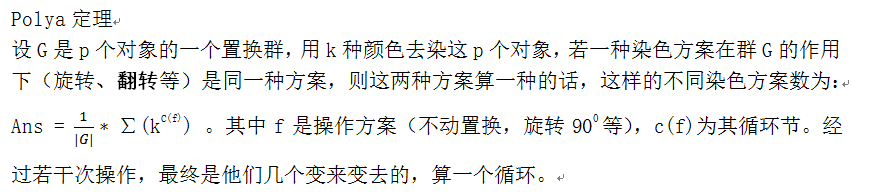
}

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***\n\n**");

}

} arr;

## 杂项



对于n个位置的项链，有n中旋转置换和n中翻转置换

对于旋转置换：

C(fi) = gcd(n,i)，i表示选择i颗宝石以后。有n种，for (i=1 to n) ans += kgcd(i,n)

对于翻转置换：

如果n是奇数，有n个置换，而且C(f) = (n + 1) / 2

如果n是偶数，有n个置换，有n / 2个 C(f) = n / 2 + 1，有n / 2个 C(f) = n / 2

下面标程：通过旋转、翻转达到同一种状态的被认为是相同的项链

LL polya(**int** k, **int** n) {

LL ans = 0;

**for** (**int** i = 1; i <= n; ++i) ans += mypow(k, gcd(i, n));

//这些都是只有一种置换，旋转一个位置，两个位置等等

**if** (n & 1) ans += mypow(k, (n + 1) / 2) \* n; //乘上n代表有n种这样的置换

**else** ans += (mypow(k, n / 2) \* (n / 2) + mypow(k, (n / 2) + 1) \* n / 2);

**return** ans / (2 \* n); //一共有2n种置换

}

附上一个枚举gcd得到的polya，n<=1e9，有n种颜色

**void** dfs(**int** cur, LL gcd, LL phi) {

**if** (cur == lena + 1) {

ans = (ans + (quick\_pow(n, gcd, p) \* phi) % p) % p;

**return** ;

}

dfs(cur + 1, gcd, phi); *//要这个*

phi \*= a[cur].pr - 1;

**for** (**int** i = 1; i <= a[cur].num; ++i) {

gcd /= a[cur].pr;

dfs (cur + 1, gcd, phi);

phi \*= a[cur].pr; *//继续不要了这个*

}

}

2、康托展开

求出1…n的全排列的第k小。（k从1开始）

假如要对一个排列进行hash，比如是排列是 D、B、A、C，那么可以hash成20，因为其在全排列中按字典序排名20。

编码：HashKeyValue = a­1­ \* 3! + a2 \* 2! + a3 \* 1! + a4 \* 0!。这样hash不会重复。其中系数a[1..n]分别代表字母a[1]（就是”D”）在**当前子集**里面是第几大，D在{A、B、C、D}中第3大，所以a[1] = 3，B在{A、B、C}中第1大，所以a[2] = 1，以此类推……..得到的值就是该排列在全排列中的字典序排名。

解释：比D小的，有3个，以这三个字母开头的排列有3!个，所以是3 \* 3!

比B小的，有1个（D已经确定在开头了），这样的排列有1 \* 2!个。

解码：辗转相除，得到HashKeyValue是第20个排列，怎么确定是D、B、A、C、？

20 / (3!) = 3，余数是2 （然后用余数继续除）

2 / (2!) = 1， 余数是0

0 / (1!) = 0，余数是0

0 / (0!) = 0，余数0

也就是，第一位，有3个数比他小，那就是D了，然后第二位，有一个数比他小，那就是B了。第三位，0个，就是A，然后剩下的，就是C。

**class** cantor { //求1...n的排列的第k大 && hash排列，

**public** : //class默认是private，所以要加上public

**int** fac[12];

cantor() { //预处理阶乘，LL的话可以处理到20！ = 2.4329020081766 \* 1018

fac[0] = 1;

**for** (**int** i = 1; i <= 11; ++i) {

fac[i] = fac[i - 1] \* i;

}

}

**int** decode(**char** str[], **int** lenstr) { //O(n \* n)的hash

**int** ans = 0;

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

**int** cnt = 0;

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

**if** (str[i] > str[j]) {

cnt++;

}

}

ans += cnt \* fac[lenstr - i];

}

**return** ans + 1;

}

vector<**int**> encode(**int** lenstr, **int** k) { //字典序排第k的是那个，k从1开始

vector<**int**>ans;

**int** toans;

**bool** vis[12] = {0};

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

**int** t = k / fac[lenstr - i];

k %= fac[lenstr - i];

**for** (toans = 1; toans <= lenstr; ++toans) {

**if** (vis[toans]) **continue**;

**if** (t == 0) **break**;

t--;

}

ans.push\_back(toans);

vis[toans] = **true**;

}

**return** ans;

}

} cantor;

3、位运算：

①、判断一个数二进制相邻两位是否同时是1，需(x & (x << 1)) > 0，成立，就是。

②、判断一个数是否是2n，只需(x & (x - 1)) == 0。

③、算出一个数二进制中，1的个数：复杂度是val二进制中1的个数

**int** calc(**int** val) {

**int** ans = 0;

**while** (val) { //遇到2n就停止

val &= val - 1;

ans++;

}

**return** ans;

}

4、求解ab的最高k位，n!的最高k位。

设x = ab，则有log10 x = b \* log10 a，所以x = 10b \* log10a

那么可以把b \* log10 a等价于两个数字A + B，一个是整数部分，一个是小数部分。

所有x = 10A \* 10B，那么其实我们只关心10B，整数部分10A相当于是ab的位数，而真正有效的是10B，需要k位，则10k – 1 \* 10B取整即可。

**int** calc(LL a, LL b, **int** k) { //a^b的前k + 1位

**double** res = b \* log10(a \* 1.0) - (LL)(b \* log10(a \* 1.0)); //获得小数部分

**return** (**int**)pow(10.0, k + res);

}

其实还可以取前15位左右来乘，这样精度也是足够的。

比如我算最高位，那么1234我转化成1.234 \* 1.234的话，和1234 \* 1234最高位的结果是一样的，用的是double能存15位左右的小数的技巧。而不至于1015 \* 1015爆long long

**int** calc(**double** a, LL b, **int** k) {

**double** ans = 1, base = a;

**while** (b) {

**if** (b & 1) ans = ans \* base;

b >>= 1;

base = base \* base;

**while** (ans >= 1000) ans /= 10; //保留前3位

**while** (base >= 1000) base /= 10;

}

**return** (**int**)ans;

}

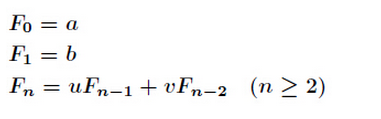
求n!的最高位也是一样，不过这里可能需要用到n！的斯特林公式

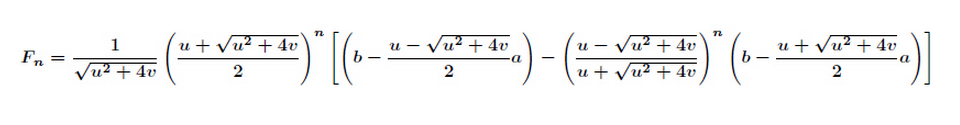
设log10x = log10(n!)，则x = 10log10(n!)，那么也是等价定义成10A \* 10B，其中我们只关心10B，也就是小数部分。那么怎么求log10(n!)的小数部分呢？

如果n比较小，则log10(n!) = log10(1) + log10(2) + ….. + log10(n)，这是很精确的公式。

如果n比较大，则有n! ≈ nn / en \* sqrt(2 \* PI \* n)，这个公式在n很大的时候，才精确。

5、广义的斐波那契数列





注意当n很大的时候，比如大于200，有些部分会趋向于0。这部分。

①、不能用系统自带的pow函数，因为它的定义是double类型的，不能保证精度。如果要算小数的pow，那么用这个刚好（精度比较好）。否则，请自己写一个pow函数。

②、解方程a\*k³<=n这样有多少组解，可以先放缩成k³<=n，然后枚举k，然后k的上限是n开3次方。现在我们已经得到了k，所求的就是a满足的个数了，a = ( n / k3 向下取整 )

③、求有n!的或者an之类的函数，可以取ln来做。取ln，是log(val)，其他的是log10(val)，和log2(val)。求阶乘有一条公式：ln(n!) ≈ (n \* ln(n)) - n + (1 / 2 \* ln(2 \* PI \* n));

④、所有大于等于6的质数，都可以表达成6 \* n – 1或者是6 \* n + 1的形式。

⑤、函数exp(1.0)就表示自然底数e1。

# 其他

## 1、STL

**1、set和multiset**

一、set:

元素集合，里面的元素自动去重，只会出现一次。而且默认保持升序排列。

**插入：set.insert(12)**

①、得到集合元素个数： book.size();

②、判断集合是否有这个元素: book.find(val) != book.end();

③、清空一个集合： book.clear(); //清空后可以继续使用

④、判断集合是否为空： book.empty();

⑤、删除一个元素 book.erase(val); //删除迭代器比较好

⑥、在set中放结构体，写个比较函数，可以按照规定顺序排列，不写不行。

struct data

{

int num;

char name[maxn];

};

struct cmp

{

bool operator () (struct data a,struct data b)

return strcmp(a.name,b.name)<0;//字典序

}; **//这个分号不能少。如果想set不去错重，则每个成员都应该出现在比较函数中**

void work ()

{

set<struct data,cmp>book;//按照比较函数排列，整数要降序，也要学这样写。

struct data a; a.num=13; strcpy(a.name,"aa"); book.insert(a);

a.num=1314; strcpy(a.name,"a"); book.insert(a);

for (set<struct data>::iterator it=book.begin();it!=book.end();it++) //iterator不用比较函数

printf ("%d %s\n",it->num,it->name); //这里值就是用他们的成员名字即可

return ;

}

输出:

1314 a

13 aa

⑦、关于set中的lower\_bound 和 book.begin()和book.end()

Set中自带的lower\_bound，返回的是>=val中的第一个元素的位置，但是我们又不能像普通的lower\_bound(a+1,a+1+n,val)-a;这样，能通过指针得到偏移地址，从而得到那个位置。这里的set中的，只能够得到值。

set<int>::iterator it=book.lower\_bound(val); 然后只能\*it，得到那个元素的值。

注意一些细节，关于越界的问题。就是，如果找不到那个元素的话，我们可以分成两类：

1、所找的元素是所有数中最小的，例如元素是{2、4}，那么你找的是1的话，返回的是book.begin();同时，如果你将book.begin();减减了，去到的是book.end(); ！！不要以为book.end();是最后一个元素，非也非也，输出它，你会得到它是有多少个元素，就是元素的个数。

2、然后如果所找元素是所有数最大的话，返回的是book.end(); **然而这是不存在的元素**

3、可以用it == book.begin()这样判断是否第一个，适当it++ 和 it--

二、multiset: 元素能重复，其他的和set一样用法。**但是删除2的话，会把所有2都删除**

所以需要用迭代器指向它，然后ss.erase(it); 就能删除一个2的。

**2、priority\_queue**

1、que.size(); //得到堆中有多少个元素

2、que.top(); //取得堆中的第一个元素，对应的普通队列，是que.front(); 注意~

3、que.pop(); //弹出堆中的第一个元素。也就是上面那个是没删除功能的。

如果堆中的元素是自己定义的结构体的话，就要写一个cmp**结构体**，放进去，进行优先级排序。

注意在STL中，要排序，除了sort是写一个cmp **函数的，**其他都是写一个cmp**结构体。**

priority\_queue<int>que; //默认是一个最大堆

priority\_queue<int,vector<int>,greater<int> >que; //最小堆，记得最后那个> >中间有个空格

struct data

{

int val;

int depend;

};

struct cmp

{

bool operator () (struct data a,struct data b)

return a.depend<b.depend;//优先级大的在前 //优先队列就是这样相反的

}; //分号不能忘记

priority\_queue<struct data,vector<struct data>,cmp>que; //这样就是自己定义的一个堆了。

**3、lower\_bound & upper\_bound**

lower\_bound : 返回数组里第一个 元素值 >= val 的pos

upper\_bound : 返回数组里第一个 元素值 > val的pos。 **找不到返回最后一个元素的下一个**

int pos = lower\_bound(a+1,a+1+n,val) – a;

**4、vector<int>a**

插入：a.push\_back(4); 取最后一个：printf ("%d\n",a.back()); 删除最后一个：a.pop\_back();

**5、倒置**：reverse(a.begin(),a.end()); **//这个倒置函数那里都可以用，甚至可以倒置char[]。**

**6、merge函数：**把两个有序数组结合

**int** a[] = {0, 1, 2, 3, 4}; **int** b[] = {0, -1, 0, 3, 66}; **int** lena = 4, lenb = 4, c[222];

**merge**(a + 1, a + 1 + lena, b + 1, b + 1 + lenb, c + 1);

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

cout << c[i] << " ";

}

如果用在vector<int>里面，那么第三个vector需要先V.resize(SIZE)，先分配好空间。

## 2、常量定义 & 手动开栈 & C++取消同步 & int范围

**常量定义：**

1、#define inf (0x3f3f3f3f)

2、const double PI = acos(-1.0); 定义PI要用这个，记得是double

3、memset设置inf ： memset(dp, **0x3f**, sizeof dp); //只有一个f 且**不能用于double**

4、memset设置负inf：memset(dp, **-0x3f**, sizeof dp); //其值是：**-1044266559**

5、ASCII可见字符包括从33~126的字符，0~32 和127均为不可见字符（控制字符和换行，空格之类的）

**手动开栈**：#pragma comment(linker,”/STACK:1024000000,1024000000”)

**取消同步：**后不能混用scanf和cin之类的 ios::sync\_with\_stdio(false);

**各种类型的范围：**

unsigned int 0～4294967295 (10位数，4e9)

int -2147483648～2147483647 (10位数，2e9 2^31 - 1) **263 - 1**

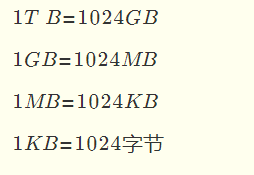
long long： -9223372036854775808～9223372036854775807 (19位数， 9e18 )

unsigned long long：0～18446744073709551615 (20位数，1e19) **264 – 1**

用unsigned long long int定义时，自然溢出相当于对264取模了。

用unsigned int的话，就相当于对232次方取模了，因为232有33位。所以是保留了最低32位**。保留最低31位的话，可以MOD = 1LL << 31，最低31位是：20 + 21 + … + 230 = 231 - 1**

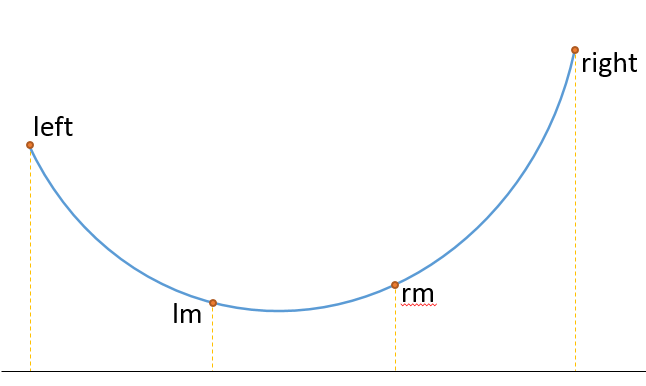
而232 – 1是有32位，也就是所有位全部是1。



那么，106的数组就需要4 \* 106个字节，就是4 \* 103个KB，就是4MB而已。

直接除以10­6，如果int就有常数4，就能知道占用多少MB。

## 3、三分答案



但当函数是凸形函数时，二分法就无法适用，这时就需要用到三分法。从三分法的名字中我们可以猜到，三分法是对于需要逼近的区间做三等分：我们发现lm这个点比rm要低，那么我们要找的最小点一定在[left,rm]之间。如果最低点在[rm,right]之间，就会出现在rm左右都有比他低的点，这显然是不可能的。 同理，当rm比lm低时，最低点一定在[lm,right]的区间内。利用这个性质，我们就可以在缩小区间的同时向目标点逼近，从而得到极值。

例题：hihocoder 1142

题意：给定一条抛物线和一个点p，要求找到点p到抛物线的最短距离。p(x,y)

**double** eps = 1e-12;

**while** (begin + eps < end) { //注意这里的区间，要和题目的一样[-200,200]

**double** Lmid = begin + (end - begin) / 3;

**double** Rmid = end - (end - begin) / 3;

**double** ret1 = dis(Lmid, f(Lmid), x, y); //计算这两个点的距离

**double** ret2 = dis(Rmid, f(Rmid), x, y); //f(x) = a\*x\*x + b\*x + c

**if** (ret1 < ret2)

end = Rmid;

**else** begin = Lmid;

}

因为begin和end无限逼近了，所以这两个是一样的。

## 4、最长下降子序列

**int** get\_pos(**int** b[], **int** lenb, **int** val) {

**int** begin = 1, end = lenb;

**while** (begin <= end) {

**int** mid = (begin + end) / 2;

**if** (b[mid] > val) begin = mid + 1;

**else** end = mid - 1;

}

**return** begin;

}

**int** dp\_down(**int** a[], **int** lena) {

**int** lenb = 0;

b[++lenb] = a[1];

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

**if** (a[i] < b[lenb]) b[++lenb] = a[i]; //不降子序列，这里要取等

**else** {

**int** pos = get\_pos(b, lenb, a[i]);

b[pos] = a[i];

}

}

**return** lenb;

}

## 5、高精度、输入挂、java大数

**1、高精度，C语言版**

**void** bigadd(**char** str1[], **char** str2[], **char** str3[]) { //str1 + str2 = str3

**int** len1 = strlen(str1 + 1), len2 = strlen(str2 + 1);

**char** b[maxn] = {0}; //maxn关键，栈分配，系统帮你释放，要时间，不乱开

**int** i = len1, j = len2;

**int** h = 1;

**while** (i >= 1 && j >= 1) b[h++] = str1[i--] - '0' + str2[j--] - '0';

**while** (i >= 1) b[h++] = str1[i--] - '0';

**while** (j >= 1) b[h++] = str2[j--] - '0';

**for** (**int** i = 1; i < h; i++) { //h是理论越界的

**if** (b[i] >= 10) {

b[i + 1]++;

b[i] -= 10;

}

}

**if** (!b[h]) --h;//没有进位到越界位。

**int** t = h;

**for** (**int** i = 1; i <= h; i++) str3[t--] = b[i] + '0';

str3[h + 1] = '\0'; //一定要手动添加结束符，不然会GG

**return** ;

}

bigadd(str1, str2, ans);

**void** bigplus(**char** str1[], **char** str2[], **char** str3[]) {

**int** len1 = strlen(str1 + 1);

**int** len2 = strlen(str2 + 1);

**int** i = 1, j = len1;

**while** (i <= j) { //倒叙

**char** ch = str1[i];

str1[i] = str1[j];

str1[j] = ch;

i++;

j--;

}

i = 1;

j = len2;

**while** (i <= j) {

**char** ch = str2[i];

str2[i] = str2[j];

str2[j] = ch;

i++;

j--;

}

**int** b[maxn] = {0}; //maxn关键，栈分配，系统帮你释放，要时间

**int** lsum = 0;

**for** (i = 1; i <= len1; i++) {

**for** (j = 1, lsum = i; j <= len2; j++, lsum++) {

b[lsum] += (str1[i] - '0') \* (str2[j] - '0');

}

}

**for** (**int** i = 1; i < lsum; i++) { // lsum 越界 lsum=len2+1

**if** (b[i] >= 10) {

b[i + 1] += b[i] / 10;

b[i] %= 10;

}

}

**if** (!b[lsum]) {

lsum--;

}

**int** h = lsum;

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

str3[h--] = b[i] + '0';

}

str3[lsum + 1] = '\0';

}

**2、输入挂**

**template** <**class** T>

**inline** **bool** fast\_in(T &ret) { //适用于正负整数 (int, long long, float, double)

**char** c;

**int** sgn;

**if** (c = getchar(), c == EOF) **return** 0; //EOF

**while** (c != '-' && (c < '0' || c > '9')) c = getchar();

sgn = (c == '-') ? -1 : 1;

ret = (c == '-') ? 0 : (c - '0');

**while** (c = getchar(), c >= '0' && c <= '9') ret = ret \* 10 + (c - '0');

ret \*= sgn;

**return** 1;

}

**3、Java大数 && Java如何做题**

求解：x­1 + x2 + x3 + …. + xn = m，其中xi 属于[Li, Ri]的不同解的个数。n <= 8

import java.util.\*; //180ms

import java.math.\*; //大数头文件

public class Main {

static final int maxn = 15; //这个也要static，不然下面无法引用类属成员。

static int[] be = new int[maxn];

static int[] en = new int[maxn];

static int n, m;

static BigInteger ans;

public static void main(String[] args) {

Scanner input = new Scanner(System.in);

int t = input.nextInt();

while ((t--) > 0) { //返回值必须bool

n = input.nextInt();

m = input.nextInt();

int sum = 0;

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

be[i] = input.nextInt();

en[i] = input.nextInt();

sum += be[i];

}

ans = BigInteger.ZERO; //清0

dfs(1, sum, 0);

System.out.println(ans);

}

}

public static BigInteger C(int n, int m) { //这个数字很大，爆LL

if (n < m) return BigInteger.ZERO; //大数的常数定义，有0, 1，和10

if (n == m) return BigInteger.ONE;

BigInteger ans = BigInteger.ONE;

int mx = Math.max(n - m, m); //调用最大值函数

int mi = n - mx;

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

ans = ans.multiply(BigInteger.valueOf(mx + i)); //转换成大数的方法

ans = ans.divide(BigInteger.valueOf(i)); //记得接收返回值

}

return ans;

}

public static void dfs(int cur, int tot, int has) {

if (cur == n + 1) {

if (has % 2 == 1) {

ans = ans.subtract(C(m - tot + n - 1, n - 1));

} else ans = ans.add(C(m - tot + n - 1, n - 1));

return;

}

dfs(cur + 1, tot - be[cur] + en[cur] + 1, has + 1); //他自己能包括en[cur]，是闭区间

dfs(cur + 1, tot, has);

}

}

## 6、基本思考方式

1、逆向思维，从结论开始递推上去。

①、缩小等边三角形（cf 712C）。可以从最终结果开始贪心模拟上去。

②、切割开销（POJ 3253），可以从最后最小的两个，进行huffman算法的合并。

③、每个i，向右边找一个符合的数。其实相当于每个i，向左边找一个符合的数。

④、循环数组最大字段，可以用总和sum – 最小字段，就解决了循环的问题。

2、有关两个变量同时限制的题目。基本思路就是排除一个变量的影响

①、HDU 4366，被权值和忠诚限制，那么可以查找时**确保权值肯定成立**。就没影响了。具体做法是对权值排序，然后从权值大的开始预处理，然后把这个人更新到线段树中。

②、GYM 101064求数组两两相加，排序后的第n小。容易想到二分答案 + 判定。判定的时候，可以枚举每一个数，**然后固定它**，去数组中（[i + 1, n]）找有多少个数相加小于等于二分值val – a[i]。其实就是固定了一个点，使得不会重复计数。

3、多列公式，进行化简。

①、POJ 2002。有多少个正方形，根据垂直和边长相等，即可枚举另外两个点。

②、CF 740D。列到的公式**同一变量放在一起**。dis[v] – dis[u] <= a[v]即是dis[v] – a[v] <= dis[u]

③、109的推公式题目，尝试下O( sqrt( 109 ) )的暴力。分类下什么情况可以这样暴力。

4、转换题目意义

①、CF 602D。区间中最大值作为这个区间的贡献值，那么所有子区间的贡献值之和是多少？

如果每个区间都扫一次，虽然可以O（1）算出最大值，但是也徒劳。把问题转化为，枚举整个数组，看看最大值是a[i]的时候有多少个区间。这个就可以用单调栈预处理出来。注意两个数值相等的情况。不能重复计数。

5、分类讨论的思想

HUST 1698，分类成 奇数 + 奇数 + 奇数 + 奇数，等的所有情况。然后还有，将奇数表达成2 \* n - 1，是一个很好的选择。

6、映射hash

HDU 1430，将起点映射到”12345678”，起点若变化，终点也跟着起点的变化来变。比如起点的6变成了3，那么应该在终点找到6，变成3。这样就把起点固定了。可以打表。然后Cantor展开。

## 杂项

1、在DFS和BFS的时候，要注意起点和终点相同，我会提前vis[]了，然后返回不了。

2、多case的题目，最好加个init();先，清空的东西都放在那里。

3、C++提交，时间比较快，内存比较大。而G++提交则相反。还有**register int**这样更快。

