## 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。

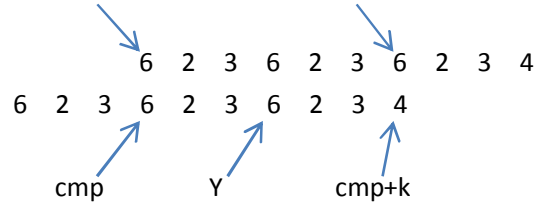
1、朴素的比较算法，设一个指针为 ans，一个为比较指针 cmp，找到 str[(ans + k) % lenstr] != str[(cmp + k) % lenstr],如果前者较大，则 ans = cmp，表明在 cmp 开始的字符串字典序更小，一路下去。复杂度为 $O(n^2)$。这里的缺陷就是，每次比较完后，ans 指针的移动量太小了，才移动到去 cmp 那里，最坏情况下，只是一格一格移动的而已，而每次移动都要比较整个串的长度。例如:bbbbbbba

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




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。

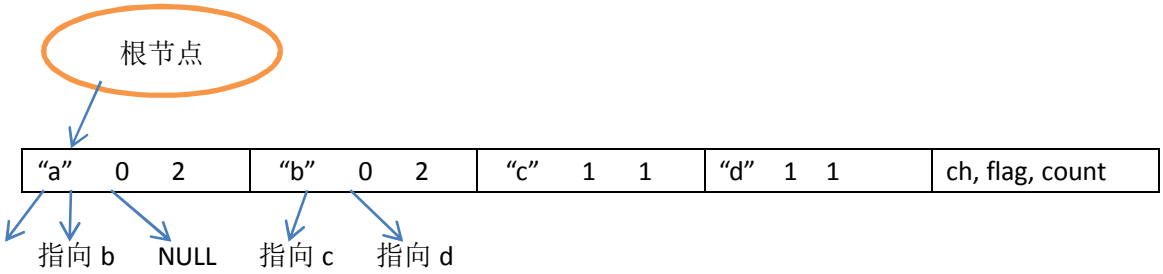
```
while (str[i+p[i]]==str[i-p[i]] && num[i-p[i]]<=num[i-p[i]+2]) ++p[i];
```

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~~




```
//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 $O(n^2)$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。

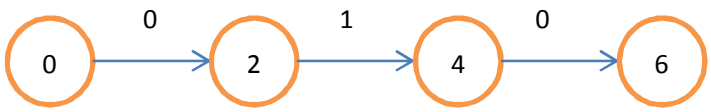
```
#include <cstdio>
#include <cstdlib>
const int maxn = 5000 + 20;
int f[maxn];
int rank[maxn];
void init () {
    for (int i = 0; i < maxn; i++) {
        f[i] = i;
        rank[i] = 0; //=0 表示奇偶性相同
    }
    return;
}
int find (int u) {
    if (u == f[u]) {
        return u;
    } else {
        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
5
1 2 even
3 4 odd
5 6 even
1 6 even
7 10 odd

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




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，2^20 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; // 10^i
    }
    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 && 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。
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。**

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 $O(n^3)$，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：$\frac{n(n-1)}{2}$ 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 $O(n^2)$，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 $O(n^2)$。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 $N^2$，这个时候比 $O(n^2)$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($n^3$ / 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（$n^2$）

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 $O(n^2)$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 $2^i$ 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**

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;                    ★、就已经标记了，不用再标记
                }
            } else if (i != father) {
                low[cur] = min(low[cur],DFN[i]);//拜师中
            }
        }
    }
}
```

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。

## 5、二分图匹配

二分图最大匹配：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
    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() {
    memset (match, 0, sizeof(match)); //多 case 这个记得 memset
    memset (book, 0, sizeof(book));
    memset (a, 0, sizeof(a));
```

```
    memset (b, 0, sizeof(b));
    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。~~**

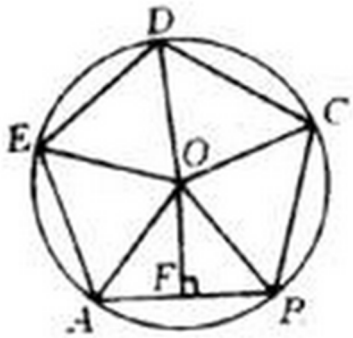
**POJ 1556　The Doors**

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;
}
```

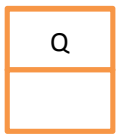
内角：
正 n 边形的内角和度数为：(n-2)×180°;
正 n 边形的一个内角是(n-2)×180°÷ n.
外角：
正 n 边形外角和等于 n*180° — (n-2)*180°=360°
所以正 n 边形的一个外角为：360°÷ n。
所以正 n 边形的一个内角也可以用这个公式：180°-360°÷ n.。
中心角：
外接圆，每条边所对的圆周角，360/n。这个角和内角是互补的
对角线：
对角线数目：**n*(n-3)/2;**
最长对角线长度

```
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 $n^2$ 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。

K 


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 $C_n^m$ 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、$C_{n-1}^{m-1} + C_{n-1}^{m} = C_n^m$
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 $C_n^m = \frac{n!}{(n-m)!*m!} = \frac{n(n-1)(n-2)\dots(n-m+1)}{m(m-1)(m-2)\dots 1}$

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（$\log_p$（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 $p_1^a * p_2^b .... * p_z^z$ 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 $p_1^a * p_2^b ....* p_k^z$ 这样的形式，然后分别求解 C(n, m) % $p_i^x$ 后，得到了模数是 $p_i^x$，余数是 C(n, m) % $p_i^x$ 的同余方程组。由于每个 $p_i^x$ 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 $p_i^x$ 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! % $p_i^x$ 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! % $p_i^x$ 的值，是会把 $p_i$ 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! % $3^2$，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）% $3^2$

等价于（1 * 2 * 4 * 5 * 7 * 8 * 10 * 11 * 13 * 14 * 16 * 17 * 19）* $3^6$ * (1 * 2 * 3 * 4 * 5 * 6) % $3^2$

然后注意到后面的，是 n / $p_i$ 的阶乘，所以这里可以递归求解。前面的，是有循环节的。每 $p_i^x$ 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% $p_i^x$ 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 $p_i^x$ 个，剩下的就暴力可以了，复杂度不大。不会超过 $p_i^x$ 个。最后，就是 $3^6$ 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 $3^8$。做法是把 n!中的 $p_i$ 因子全部提取出来另外算，这样也使得 n! % $p_i^x$ 后，和 $p_i^x$ 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,$\sqrt{n}$]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$\sqrt{n}$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**

$$C = AB = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix} = \begin{pmatrix} 1\times1+2\times2+3\times3 & 1\times4+2\times5+3\times6 \\ 4\times1+5\times2+6\times3 & 4\times4+5\times5+6\times6 \end{pmatrix} = \begin{pmatrix} 14 & 32 \\ 32 & 77 \end{pmatrix}$$

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。$X^2 – (x - 1)^2 = 2 * x + 1$
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 $2^n - 1$ 是一个质数，那么，由公式 $N(n)=2^{(n - 1)} * (2^n - 1)$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 $\frac{ans1}{ans2}\%MOD$ 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：
$x = k_1 * MOD[1] + r[1]$
$x = k_2 * MOD[2] + r[2]$
合并起来，就是：$k_1 * MOD[1] - k_2 * MOD[2] = r[2] - r[1]$
可以用扩展欧几里德算法求出 $k_1$，那么这时候。$x = k_1 * MOD[1] + r[1]$，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：$4k_1 - 2k_2 = -2$ 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，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));

老大，我不想求逆元。可以：(这个不是逆元的概念，且要注意前提，mod 后不能重复使用)
这个黑科技不要求 b 和 mod 互质哦!!!

若 b|a (这个是前提)， 则 $(\frac{a}{b})\%p$ <--> $\frac{a\%(b*p)}{b}$

证明：设 ans=( $\frac{a}{b}$ )%p --> ( $\frac{a}{b}$ ) = k*p+ans( 余数 ) --> a=kbp+b*ans; 同时 mod(bp)-->amod(bp)=b*ans.这里 kbp 相当于 k*(bp),b*ans 是余数。所以有, ans= $\frac{a\%(b*p)}{b}$

## 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) = $\frac{C(2*n,n)}{n+1}$ //如果 MOD 的数不是质数，就用递推公式。O($n^2$)

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、$\sum_{d|n} phi(d) = n$**

```
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：$C_n^1 + C_n^2 + .... + C_n^n = 2^n - C_n^0$。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 $2^q$ 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 $2^q$ 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[$primeMax^2$,$primeMax^2$]。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内积分的值，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 $0 * X_1 + 0 * X_2 + .... + 0 * X_n = val$（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(n^3)
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) {
                    a[i][j] -= a[i][col] * a[k][j];
                }
                a[i][col] = 0.0;
            }
//              debug();
        }
        for (k; k <= equ; ++k) {
            if (!same(a[k][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

Polya 定理

设 G 是 p 个对象的一个置换群，用 k 种颜色去染这 p 个对象，若一种染色方案在群 G 的作用下（旋转、翻转等）是同一种方案，则这两种方案算一种的话，这样的不同染色方案数为：

$Ans = \frac{1}{|G|} * \Sigma(k^{c(f)})$ 。其中 f 是操作方案（不动置换，旋转 $90^0$ 等），c(f)为其循环节。经过若干次操作，最终是他们几个变来变去的，算一个循环。

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：

$C(f_i) = gcd(n,i)$，i 表示选择 i 颗宝石以后。有 n 种，for (i=1 to n) ans += $k^{gcd(i,n)}$

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 / 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 = $a_1$ * 3! + $a_2$ * 2! + $a_3$ * 1! + $a_4$ * 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 * 10^18
        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 $2^n$，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 2^n 就停止
        val &= val - 1;
        ans++;
    }
    return ans;
}
```

4、求解 $a^b$ 的最高 k 位，n!的最高 k 位。

设 $x = a^b$，则有 $\log_{10} x = b * \log_{10} a$，所以 $x = 10^{b * \log_{10} a}$

那么可以把 $b * \log_{10} a$ 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 $x = 10^A * 10^B$，那么其实我们只关心 $10^B$，整数部分 $10^A$ 相当于是 $a^b$ 的位数，而真正有效的是 $10^B$，需要 k 位，则 $10^{k-1} * 10^B$ 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 $10^{15} * 10^{15}$ 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 $\log_{10}x = \log_{10}(n!)$，则 $x = 10^{\log_{10}(n!)}$，那么也是等价定义成 $10^A * 10^B$，其中我们只关心 $10^B$，也就是小数部分。那么怎么求 $\log_{10}(n!)$的小数部分呢？
如果 n 比较小，则 $\log_{10}(n!) = \log_{10}(1) + \log_{10}(2) + ..... + \log_{10}(n)$，这是很精确的公式。
如果 n 比较大，则有 $n! \approx n^n / e^n * \sqrt{2 * PI * n}$，这个公式在 n 很大的时候，才精确。

5、广义的斐波那契数列

$$F_0 = a$$
$$F_1 = b$$
$$F_n = uF_{n-1} + vF_{n-2} \quad (n \geq 2)$$

$$F_n = \frac{1}{\sqrt{u^2+4v}}\left(\frac{u+\sqrt{u^2+4v}}{2}\right)^n\left[\left(b-\frac{u-\sqrt{u^2+4v}}{2}a\right)-\left(\frac{u-\sqrt{u^2+4v}}{u+\sqrt{u^2+4v}}\right)^n\left(b-\frac{u+\sqrt{u^2+4v}}{2}a\right)\right]$$

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 $a*k^3 <= n$ 这样有多少组解，可以先放缩成 $k^3 <= n$，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，$a = ( n / k^3$ 向下取整 )

③、求有 n!的或者 $a^n$ 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：$\ln(n!) \approx (n * \ln(n)) - n + (1 / 2 * \ln(2 * PI * n))$;
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 $e^1$。

# 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) $2^{63}-1$

long long： -9223372036854775808～9223372036854775807 (19 位数， 9e18 )

unsigned long long：0～18446744073709551615 (20 位数，1e19) $2^{64}-1$

用 unsigned long long int 定义时，自然溢出相当于对 $2^{64}$ 取模了。

用 unsigned int 的话，就相当于对 $2^{32}$ 次方取模了，因为 $2^{32}$ 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：$2^0 + 2^1 + \ldots + 2^{30} = 2^{31} - 1$**

而 $2^{32} - 1$ 是有 32 位，也就是所有位全部是 1。

$1TB=1024GB$

$1GB=1024MB$

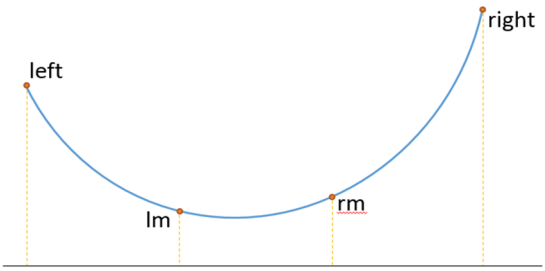
$1MB=1024KB$

$1KB=1024$字节

那么，$10^6$ 的数组就需要 4 * $10^6$ 个字节，就是 4 * $10^3$ 个 KB，就是 4MB 而已。

直接除以 $10^6$，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：$x_1 + x_2 + x_3 + .... + x_n = m$，其中 $x_i$ 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、$10^9$ 的推公式题目，尝试下 O( sqrt( $10^9$ ) )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。

| double f; | G++提交 | C++提交 | 本机gcc测试 | 最安全的方法 |
|---|---|---|---|---|
| 输入 | scanf("%lf", &f); | scanf("%lf", &f); | scanf("%lf", &f); | cin >> f; |
| 输出 | printf("%f", f); | printf("%lf", f); | printf("%lf", f); | cout << f; |