ACM 数学模板

Shen

September 10, 2014

Contents

1 全局

1.1 输入输出外挂

用于加速整数的输入输出，主要应对较大的数据量。

```
//适用于正负整数
template <class T> inline bool scan_d(T &ret)
{
    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;
}

inline void out(int x)
{
    if (x > 9) out(x / 10); putchar(x % 10 + '0');
}
```

1.2 高精度

一般最好使用 Java，因为可以省去大量的码代码的时间。不到万不得已不要用完全大数模板。

1.2.1 一般大数模板

轻量级大数模板，根据需要添加函数。

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <cstdlib>
using namespace std;

#define MAXN 9999
#define MAXSIZE 1010
#define DLEN 4
class BigNum
{
private:
    int len, a[500];
public:
    BigNum() { len = 1; memset(a, 0, sizeof(a)); }
    BigNum(const int);
    BigNum(const char*);
    BigNum(const BigNum &);
    BigNum &operator=(const BigNum &);
    friend istream& operator>>(istream&, BigNum&);
    friend ostream& operator<<(ostream&, BigNum&);
    BigNum operator+(const BigNum &) const;
    BigNum operator-(const BigNum &) const;
    BigNum operator*(const BigNum &) const;
    BigNum operator/(const int &) const;
    BigNum operator^(const int &) const;
    int operator%(const int &) const;
    bool operator>(const BigNum &T) const;
    bool operator>(const int &t) const;
    void print();
};

BigNum::BigNum(const int b)
```

```
{
    int c, d = b;
    len = 0; memset(a, 0, sizeof(a));
    while (d > MAXN)
    {
        c = d - (d / (MAXN + 1)) * (MAXN + 1);
        d = d / (MAXN + 1);
        a[len++] = c;
    }
    a[len++] = d;
}

BigNum::BigNum(const char *s)
{
    int t, k, index, L, i;
    memset(a, 0, sizeof(a));
    L = strlen(s);
    len = L / DLEN;
    if (L % DLEN) len++;
    index = 0;
    for (i = L - 1; i >= 0; i -= DLEN)
    {
        t = 0;
        k = i - DLEN + 1;
        if (k < 0) k = 0;
        for (int j = k; j <= i; j++)
            t = t * 10 + s[j] - '0';
        a[index++] = t;
    }
}
BigNum::BigNum(const BigNum &T):len(T.len)
{
    memset(a, 0, sizeof(a));
    for (int i = 0; i < len; i++)
        a[i] = T.a[i];
}
BigNum & BigNum::operator=(const BigNum &n)
{
    len = n.len;
    memset(a, 0, sizeof(a));
    for (int i = 0; i < len; i++)
        a[i] = n.a[i];
    return *this;
}
istream& operator>>(istream &in, BigNum &b)
{
    char ch[MAXSIZE * 4];
    int i = -1;
    in >> ch;
    int L = strlen(ch);
    int count = 0, sum = 0;
    for (i = L - 1; i >= 0; )
    {
        sum = 0;
        int t = 1;
        for (int j = 0; j < 4 && i >= 0; j++, i--, t *= 10)
            sum += (ch[i] - '0') * t;
        b.a[count] = sum;
        count++;
    }
    b.len = count++;
    return in;
}
ostream& operator<<(ostream& out, BigNum& b)
{
    cout << b.a[b.len - 1];
    for (int i = b.len - 2; i >= 0; i--)
        printf("%04d", b.a[i]);
```

```
102     return out;
103 }
104 BigNum BigNum::operator+(const BigNum &T) const
105 {
106     BigNum t(*this);
107     int big = (T.len > len)? T.len: len;
108     for (int i = 0; i < big; i++)
109     {
110         t.a[i] += T.a[i];
111         if (t.a[i] > MAXN)
112         {
113             t.a[i + 1]++;
114             t.a[i] -= MAXN+1;
115         }
116     }
117     if (t.a[big] != 0) t.len = big + 1;
118     else t.len = big;
119     return t;
120 }
121 BigNum BigNum::operator-(const BigNum &T) const
122 {
123     bool flag;
124     BigNum t1, t2;
125     if (*this > T)
126     {
127         t1 = *this;
128         t2 = T;
129         flag = 0;
130     }
131     else
132     {
133         t1 = T;
134         t2 = *this;
135         flag = 1;
136     }
137     int big = t1.len;
138     for (int i = 0, j = 0; i < big; i++)
139     {
140         if (t1.a[i] < t2.a[i])
141         {
142             j = i + 1;
143             while (t1.a[j] == 0) j++;
144             t1.a[j--]--;
145             while (j > i) t1.a[j--] += MAXN;
146             t1.a[i] += MAXN + 1 - t2.a[i];
147         }
148         else t1.a[i] -= t2.a[i];
149     }
150     t1.len = big;
151     while (t1.a[len - 1] == 0 && t1.len > 1) { t1.len--; big--; }
152     if (flag) t1.a[big - 1] = 0 - t1.a[big - 1];
153     return t1;
154 }
155 BigNum BigNum::operator*(const BigNum &T) const
156 {
157     BigNum ret;
158     for (int i = 0, j = 0; i < len; i++)
159     {
160         int up = 0;
161         for (j = 0; j < T.len; j++)
162         {
163             int temp = a[i] * T.a[j] + ret.a[i + j] + up;
164             if (temp > MAXN)
165             {
166                 int temp1 = temp - temp / (MAXN + 1) * (MAXN + 1);
167                 up = temp / (MAXN + 1);
168                 ret.a[i + j] = temp1;
169             }
```

```
170             else
171             {
172                 up = 0;
173                 ret.a[i + j] = temp;
174             }
175         }
176         if (up != 0) ret.a[i + j] = up;
177     }
178     ret.len = len + T.len;
179     while (ret.a[ret.len - 1] == 0 && ret.len > 1) ret.len--;
180     return ret;
181 }
182 BigNum BigNum::operator/(const int &b) const
183 {
184     BigNum ret;
185     int i, down=0;
186     for (i=len-1; i>=0; i--)
187     {
188         ret.a[i]=(a[i]+down*(MAXN+1))/b;
189         down=a[i]+down*(MAXN+1)-ret.a[i]*b;
190     }
191     ret.len=len;
192     while (ret.a[ret.len-1]==0 && ret.len>1)
193         ret.len--;
194     return ret;
195 }
196 int BigNum::operator%(const int &b) const
197 {
198     int i, d=0;
199     for (i=len-1; i>=0; i--)
200         d=((d*(MAXN+1))%b+a[i])%b;
201     return d;
202 }
203 BigNum BigNum::operator^(const int &n) const
204 {
205     BigNum t, ret(1);
206     if (n < 0) exit(-1);
207     if (n == 0) return 1;
208     if (n == 1) return *this;
209     int m = n, i = 1;
210     while (m > 1)
211     {
212         t = *this;
213         for (i = 1; (i << 1) <= m; i <<= 1)
214             t = t * t;
215         m -= i;
216         ret = ret * t;
217         if (m == 1) ret = ret * (*this);
218     }
219     return ret;
220 }
221 bool BigNum::operator>(const BigNum &T) const
222 {
223     int ln;
224     if (len > T.len) return true;
225     else if (len == T.len)
226     {
227         ln = len - 1;
228         while (a[ln] == T.a[ln] && ln >= 0) ln--;
229         if (ln >= 0 && a[ln] > T.a[ln])
230             return true;
231         else
232             return false;
233     }
234     else
235         return false;
236 }
237 bool BigNum::operator>(const int &t) const
```

```
{
    BigNum b(t);
    return *this > b;
}
void BigNum::print()
{
    printf("%d", a[len - 1]);
    for (int i = len - 2; i >= 0; i--)
        printf("%04d", a[i]);
    printf("\n");
}

int main()
{
    BigNum a, b, c;
    while (cin >> a >> b)
    {
        c = a + b;
        cout << a << " + " << b << " = " << c << endl;
    }
    return 0;
}
```

1.2.2 完全大数模板

完全大数模板，根据需要添加函数。
另外该代码十分冗长，请务必注意。

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <cstdlib>
#include <sstream>
using namespace std;

class BigInt
{
    private:
        char *digits;
        int size;            // number of used bytes (digits)
        int capacity;        // size of digits
        int sign;            // -1, 0 or +1

    public:
        /** Creates a BigInt with initial value n and initial capacity cap **/
        BigInt(int n, int cap);

        /** Creates a BigInt with initial value n **/
        BigInt(int n);

        /** Creates a BigInt with initial value floor(d) **/
        BigInt(long double d);

        /** Creates a BigInt with value 0 **/
        BigInt();

        /** Creates a BigInt by reading the value from a string **/
        BigInt(string s);

        /** Creates a BigInt by reading the value from a C string **/
        BigInt(const char s[]);

        /** Copy constructor **/
        BigInt(const BigInt& n);

        /** Assignment operators **/
        const BigInt &operator=(const BigInt& n);
```

```
40          const BigInt &operator=(int n);
41
42          /** Cleans up **/
43          ~BigInt();
44
45          /** Removes any leading zeros and adjusts the sign **/
46          void normalize();
47
48          /** Returns the sign of n: -1, 0 or 1 **/
49          static int sig(int n);
50
51          /** Returns the sign of n: -1, 0 or 1 **/
52          static int sig(long double n);
53
54          /** Returns the number of decimal digits **/
55          inline int length() { return size; }
56
57          /** Arithmetic **/
58          BigInt operator-();
59          BigInt operator+ (BigInt n);
60          BigInt&operator+=(BigInt n);
61          BigInt operator- (BigInt n);
62          BigInt&operator-=(BigInt n);
63          BigInt operator* (BigInt n);
64          void   operator*=(BigInt n);
65          BigInt operator/ (BigInt n);
66          void   operator/=(BigInt n);
67          BigInt operator% (BigInt n);
68          void   operator%=(BigInt n);
69          // Divides storing quotient in *this and returning the remainder
70          int divide(int n);
71          BigInt divide(BigInt n);
72
73          /** Casting **/
74          bool operator!();
75          operator bool();
76          operator string();
77
78          /** Comparison **/
79          bool operator<(BigInt n);
80          bool operator>(BigInt n);
81          bool operator==(BigInt n);
82          bool operator<=(BigInt n);
83          bool operator>=(BigInt n);
84          int compare(BigInt n);
85
86          /** Returns the lowest value as an integer **/
87          /** (watch out for overflow) **/
88          int toInt();
89
90          /** Returns the value as a decimal string **/
91          string toString();
92
93          /** Outputs decimal value to stdout **/
94          void print();
95
96          /** Outputs the decimal value, with commas **/
97          void printWithCommas(ostream &out);
98
99      private:
100         /** Expansion **/
101         void grow();
102
103     /** I/O Friends **/
104     friend istream &operator>>(istream &in, BigInt& n);
105     friend ostream &operator<<(ostream &out, BigInt n);
106 };
107
```

```
/** Misc **/
inline bool isDigit(int c) { return(c >= (int)'0' && c <= (int)'9'); }

/** Input/Output **/
istream& operator>>(istream& in, BigInt& n)
{
    n.size = 0;
    n.sign = 1;
    int sign = 1;
    int c;
    while ((c = in.peek()) >= 0 &&
           (c == ' ' || c == '\t' || c == '\r' || c == '\n'))
        in.get();
    if (c < 0 || (c != (int)'-' && !isDigit(c))) { in >> c; return in; }
    if (c == (int)'-') { sign = -1; in.get(); }

    // FIXME: Extremely inefficient! Use a string.
    while ((c = in.peek()) >= 0 && isDigit(c))
    {
        in.get();
        n *= 10;
        n += (c - (int)'0');
    }
    n.sign = sign; // XXX: assign n.sign directly after fixing the FIXME
    n.normalize();
    return in;
}

ostream& operator<<(ostream& out, BigInt n) { return out << n.toString(); }

BigInt::BigInt(int n, int cap)
{
    cap = max(cap, (int)sizeof(n) * 8);
    capacity = cap;
    sign = sig(n);
    n *= sign;
    digits = new char[cap];
    memset(digits, 0, cap);
    for (size = 0; n; size++)
    {
        digits[size] = n % 10;
        n /= 10;
    }
}

BigInt::BigInt(int n)
{
    capacity = 1024;
    sign = sig(n);
    n *= sign;
    digits = new char[capacity];
    memset(digits, 0, capacity);
    size = 0;
    while (n)
    {
        digits[size++] = n % 10;
        n /= 10;
    }
}

BigInt::BigInt()
{
    capacity = 128;
    sign = 0;
    digits = new char[capacity];
    memset(digits, 0, capacity);
    size = 0;
}
```

```
176
177 BigInt::BigInt(string s)
178 {
179     capacity = max((int)s.size(), 16);
180     sign = 0;
181     digits = new char[capacity];
182     memset(digits, 0, capacity);
183     istringstream in(s);
184     in >> (*this);
185 }
186
187 BigInt::BigInt(const char s[])
188 {
189     capacity = max((int)strlen(s), 16);
190     sign = 0;
191     digits = new char[capacity];
192     memset(digits, 0, capacity);
193     istringstream in(s);
194     in >> (*this);
195 }
196
197 BigInt::BigInt(const BigInt& n)
198 {
199     capacity = n.capacity;
200     sign = n.sign;
201     size = n.size;
202     digits = new char[capacity];
203     memcpy(digits, n.digits, capacity);
204 }
205
206 const BigInt& BigInt::operator=(const BigInt& n)
207 {
208     if (&n != this)
209     {
210         if (capacity < n.size)
211         {
212             capacity = n.capacity;
213             delete [] digits;
214             digits = new char[capacity];
215         }
216         sign = n.sign;
217         size = n.size;
218         memcpy(digits, n.digits, size);
219         memset(digits + size, 0, capacity - size);
220     }
221     return *this;
222 }
223
224 const BigInt& BigInt::operator=(int n)
225 {
226     sign = sig(n);
227     n *= sign;
228     for (size = 0; n; size++)
229     {
230         digits[size] = n % 10;
231         n /= 10;
232     }
233     return *this;
234 }
235
236 BigInt::~BigInt() { delete [] digits; }
237
238 void BigInt::normalize()
239 {
240     while (size && !digits[size-1]) size--;
241     if (!size) sign = 0;
242 }
243
```

```
244 int BigInt::sig(int n) {return(n > 0 ? 1 : (n == 0 ? 0 : -1)); }
245
246 int BigInt::sig(long double n) { return(n > 0 ? 1 : (n == 0 ? 0 : -1)); }
247
248 string BigInt::toString()
249 {
250     string s = (sign >= 0 ? "" : "-");
251     for (int i = size - 1; i >= 0; i--)
252         s += (digits[i] + '0');
253     if (size == 0) s += '0';
254     return s;
255 }
256
257 void BigInt::print() { cout << toString(); }
258
259 void BigInt::grow()
260 {
261     char *olddigits = digits;
262     int oldCap = capacity;
263     capacity *= 2;
264     digits = new char[capacity];
265     memcpy(digits, olddigits, oldCap);
266     memset(digits + oldCap, 0, oldCap);
267     delete [] olddigits;
268 }
269
270 BigInt BigInt::operator-()
271 {
272     BigInt result(*this);
273     result.sign *= -1;
274     return result;
275 }
276
277 BigInt BigInt::operator+(BigInt n)
278 {
279     BigInt result(*this);
280     result += n;
281     return result;
282 }
283
284 BigInt &BigInt::operator+=(BigInt n)
285 {
286     int maxS = max(size, n.size) + 1;
287     while (maxS >= capacity) grow(); // FIXME: this is stupid
288     if (!n.sign) return *this;
289     if (!sign) sign = n.sign;
290     if (sign == n.sign)
291     {
292         int carry = 0;
293         int i;
294         for (i = 0; i < maxS - 1 || carry; i++)
295         {
296             int newdig = carry;
297             if (i < size) newdig += digits[i];
298             if (i < n.size) newdig += n.digits[i];
299             digits[i] = newdig % 10;
300             carry = newdig / 10;
301         }
302         size = max(i, size);
303     }
304     else
305     {
306         n.sign *= -1;
307         operator-=(n);
308         n.sign *= -1;
309     }
310     return *this;
311 }
```

```
312
313 BigInt BigInt::operator-(BigInt n)
314 {
315     BigInt result(*this);
316     result -= n;
317     return result;
318 }
319
320 BigInt &BigInt::operator-=(BigInt n)
321 {
322     int maxS = max(size, n.size) + 1;
323     while (maxS >= capacity) grow(); // FIXME: this is stupid
324     if (!n.sign) return *this;
325     if (!sign) sign = 1;
326     if (sign == n.sign)
327     {
328         if (sign >= 0 && *this < n || sign < 0 && *this > n)
329         {
330             // Subtracting a bigger number
331             BigInt tmp = n;
332             tmp -= *this;
333             *this = tmp;
334             sign = -sign;
335             return *this;
336         }
337
338         int carry = 0;
339         int i;
340         for (i = 0; i < maxS - 1; i++)
341         {
342             int newdig = carry;
343             if (i < size) newdig += digits[i];
344             if (i < n.size) newdig -= n.digits[i];
345             if (newdig < 0) newdig += 10, carry = -1;
346             else carry = 0;
347             digits[i] = newdig;
348         }
349         if (carry) // Subtracted a bigger number, need to flip sign
350         {
351             if (i) digits[0] = 10 - digits[0];
352             size = (i ? 1 : 0);
353             for (int j = 1; j < i; j++)
354             {
355                 digits[j] = 9 - digits[j];
356                 if (digits[i]) size = j + 1;
357             }
358             sign *= -1;
359         }
360         normalize();
361     }
362     else
363     {
364         n.sign *= -1;
365         operator+=(n);
366         n.sign *= -1;
367     }
368     return *this;
369 }
370
371 BigInt BigInt::operator*(BigInt n)
372 {
373     BigInt result(0, size + n.size);
374     result.sign = sign * n.sign;
375     if (!result.sign) return result;
376     int i, j;
377     for (i = 0; i < n.size; i++)
378     {
379         if (n.digits[i])
```

```
380         {
381             int carry = 0;
382             for (j = 0; j < size || carry; j++)
383             {
384                 int newDig =
385                     result.digits[i + j] +
386                     (j < size ? n.digits[i] * digits[j] : 0) +
387                     carry;
388                 result.digits[i + j] = newDig % 10;
389                 carry = newDig / 10;
390             }
391         }
392     }
393     result.size = i + j - 1;
394     return result;
395 }
396
397 void BigInt::operator*=(BigInt n)
398 {
399     operator=(operator*(n));
400 }
401
402 BigInt BigInt::operator/(BigInt n)
403 {
404     if (!n) n.size /= n.size;       //XXX: force a crash
405     BigInt result(*this);
406     result /= n;
407     return result;
408 }
409
410 void BigInt::operator/=(BigInt n){ divide(n); }
411
412 BigInt BigInt::operator%(BigInt n)
413 {
414     BigInt tmp(*this);
415     return tmp.divide(n);
416 }
417
418 BigInt BigInt::divide(BigInt n)
419 {
420     if (!n) n.size /= n.size;         //XXX: force a crash
421     if (!sign) return 0;
422     sign *= n.sign;
423     int oldSign = n.sign;
424     n.sign = 1;
425     BigInt tmp(0, size);
426     for (int i = size - 1; i >= 0; i--)
427     {
428         tmp *= 10;
429         tmp += digits[i];
430         digits[i] = 0;
431         while (tmp >= n) { tmp -= n; digits[i]++; }
432     }
433     normalize();
434     n.sign = oldSign;
435     return tmp;
436 }
437
438 bool BigInt::operator!() { return !size; }
439
440 BigInt::operator bool() { return size; }
441
442 BigInt::operator string() { return toString(); }
443
444 bool BigInt::operator<(BigInt n) { return(compare(n) < 0); }
445
446 bool BigInt::operator>(BigInt n) { return(compare(n) > 0); }
447
```

```
448 bool BigInt::operator==(BigInt n) { return(compare(n) == 0); }
449
450 bool BigInt::operator<=(BigInt n) { return(compare(n) <= 0); }
451
452 bool BigInt::operator>=(BigInt n) { return(compare(n) >= 0); }
453
454 int BigInt::compare(BigInt n)
455 {
456     if (sign < n.sign) return -1;
457     if (sign > n.sign) return 1;
458     if (size < n.size) return -sign;
459     if (size > n.size) return sign;
460     for (int i = size - 1; i >= 0; i--)
461     {
462         if (digits[i] < n.digits[i]) return -sign;
463         else if (digits[i] > n.digits[i]) return sign;
464     }
465     return 0;
466 }
467
468 int main()
469 {
470     BigInt a, b, c;
471     while (cin >> a >> b)
472     {
473         c = a + b;
474         cout << a << " + " << b << " = " << c << endl;
475     }
476     return 0;
477 }
```

1.2.3 Java BigInteger

利用 Java 来处理大数问题是一件很轻松愉快的事情，不仅写起来简单方便，而且能够省去大量的调试时间。不过运算函数最好全部写成函数名调用，尽量不要直接使用运算符调用。

数据定义与输入输出

方法	接收参数	作用
BigInteger n = BigInteger.new(val)	String, int, void	新建一个值为 val 的大数
Scanner cin = Scanner(System.in); n = cin.nextBigInteger();		读入一个大数
System.out.print(n);		输出大数 n
System.out.println(n);		输出大数 n 并且换行
System.out.printf("%d\n", n);		类似 C++ 格式化输出大数 n

基本常量与方法

方法	接收参数	作用
BigInteger.ONE		常量，值为 1
BigInteger.TEN		常量，值为 10
BigInteger.ZERO		常量，值为 0
String toString()	void	返回 10 进制下的字符串表示形式
String toString(radix)	int	返回基于 radix 进制下的字符串表示形式
BigInteger valueOf(val)	String, long, int	将 val 的值赋给 this
int compareTo(val)	BigInteger	根据小于、等于或大于 val 返回 -1，0，1
boolean equals(x)	Object	判断是否与指定的 Object 相等

运算方法

方法	接收参数	作用
BigInteger abs()	void	返回其绝对值
BigInteger negate()	void	返回其相反数
int signum()	void	返回其符号函数
BigInteger add(val)	BigInteger	返回一值为 (this + val) 的大数
subtract(val)	BigInteger	返回一值为 (this - val) 的大数
BigInteger muliply(val)	BigInteger	返回一值为 (this * val) 的大数
BigInteger divide(val)	BigInteger	返回一值为 (this / val) 的大数
BigInteger remainder(val)	BigInteger	返回一值为 (this % val) 的大数
BigInteger[] divideAndRemainder(val)	BigInteger	a[0] = this / val a[1] = this % val
BigInteger mod(val)	BigInteger	返回一值为 (this mod val) 的大数
BigInteger pow(val)	BigInteger	返回一值为 (this ^val) 的大数
BigInteger max(val)	BigInteger	返回 this, val 的最大值
BigInteger min(val)	BigInteger	返回 this, val 的最小值
BigInteger and(val)	BigInteger	返回一值为 (this and val) 的大数
BigInteger andNot(val)	BigInteger	返回一值为 (this and val) 的大数
BigInteger not(val)	BigInteger	返回一值为 (this not val) 的大数
BigInteger or(val)	BigInteger	返回一值为 (this or val) 的大数
BigInteger xor(val)	BigInteger	返回一值为 (this xor val) 的大数
BigInteger shiftLeft(n)	int	返回一值为 (this « n) 的大数
BigInteger shiftRight(n)	int	返回一值为 (this » n) 的大数

1.2.4 Java BigDecimal

BigDecimal 的基本用法与 BigInteger 大同小异，所以关于其基本方法不再赘述。在此给出两个常用的方法：

方法	接收参数	作用
BigInteger stripTrailingZeros()	void	去除后导零
String toPlainString()	void	返回非科学计数法环境下的数值

1.3 分数类

完成分数的加减乘除运算。成员变量为分子 (num) 与分母 (den)，只能通过分子分母来进行构造，并且重载了 +, -, *, /, < ,== 运算符。

```
typedef long long int64;
int64 gcd(int64 a, int64 b) { return a == 0? m: gcd(b, a % b); }
struct Fraction
{
    int64 num, den;
    Fraction(int64 n = 0, d = 0)
    {
        if (d < 0) { n = -n; d = -d; }
        assert(d != 0);
        int64 g = gcd(abs(n), d);
        num = n / g; den = d / g;
    }
    Fraction operator+(const Fraction& o) const
    {
        return Fraction(num * o.den + den * o.num, den * o.den);
    }
    Fraction operator-(const Fraction& o) const
    {
        return Fraction(num * o.den - den * o.num, den * o.den);
```

```
20      }
21      Fraction operator*(const Fraction& o) const
22      {
23          return Fraction(num * o.num, den * o.den);
24      }
25      Fraction operator/(const Fraction& o) const
26      {
27          return Fraction(num * o.den, den * o.num);
28      }
29      Fraction operator<(const Fraction& o) const
30      {
31          return num * o.den < den * o.num;
32      }
33      Fraction operator==(const Fraction& o) const
34      {
35          return num * o.den == den * o.num;
36      }
37  };
```

2 矩阵

2.1 矩阵类

实现矩阵的基础计算，通过控制全局变量来控制矩阵的大小。使用前务必清零，即调用 clear() 方法

```
const int MaxN = 1010;
const int MaxM = 1010;
struct Matrix
{
    int n, m;
    int a[MaxN][MaxM];
    void clear() { n = m = 0; memset(a, 0, sizeof(a)); }
    Matrix operator+(const Matrix& b) const
    {
        Matrix tmp; tmp.n = n; tmp.m = m;
        for (int i = 0; i < n; i++)
            for (int j = 0; j < m; j++)
                tmp.a[i][j] = a[i][j] + b.a[i][j];
        return tmp;
    }
    Matrix operator-(const Matrix& b) const
    {
        Matrix tmp; tmp.n = n; tmp.m = m;
        for (int i = 0; i < n; i++)
            for (int j = 0; j < m; j++)
                tmp.a[i][j] = a[i][j] - b.a[i][j];
        return tmp;
    }
    Matrix operator*(const Matrix& b) const
    {
        Matrix tmp; tmp.clear(); tmp.n = n; tmp.m = b.m;
        for (int i = 0; i < n; i++)
            for (int j = 0; j < b.m; j++)
                for (int k = 0; k < m; k++)
                    tmp.a[i][j] += a[i][k] * b.a[k][j];
        return tmp;
    }
    void print()
    {
        printf("n = %d, m = %d\n", n, m);
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < m; j++)
                printf("%4d", a[i][j]);
            puts("");
        }
        puts("");
    }
};
```

2.2 Gauss 消元

给出一个 n 元一次方程组，求他们的解集。
将这个方程组变换成矩阵形式，利用初等变换得到上三角矩阵，最后回代得到解集。

复杂度	$O(n^3)$	
输入	a	方程组对应的矩阵
	n	未知数的个数
	l, ans	存储解，l[] 表示是否为自由元
输出		解空间的维数

```
const int MaxN = 105;
const double EPS = 1e-8;
```

```
inline int solve(double a[][MaxN], bool l[], double ans[], const int& n)
{
    // old format: A[][] * x[] = B[]
    // new format: A[][0 .. n - 1] * x[] = A[][n]
    // the last row is B[] so called
    int res = 0, r = 0;
    for (int i = 0; i < n; i++) l[i] = false;
    for (int i = 0; i < n; i++)
    {
        for (int j = r; j < n; j++) if (fabs(a[j][i] > EPS)
        {
            for (int k = i; k <= n; k++) swap(a[j][k], a[r][k]);
            break;
        }
        if (fabs(a[r][i] < EPS) { res++; continue; }
        for (int j = 0; j < n; j++)
            if (j != r && fabs(a[j][i] > EPS)
        {
            double tmp = a[j][i] / a[r][i];
            for (int k = i; k <= n; k++) a[j][k] -= tmp * a[r][k];
        }
        l[i] = true; r++;
    }
    for (int i = 0; i < n; i++) if (l[i])
        for ( int j = 0; j < n; j++) if (fabs(a[j][i]) > 0)
            ans[i] = a[j][n] / a[j][i];
    return res;
}
```

2.3 矩阵的逆

给一个 n 阶矩阵，求它的逆。

将原矩阵 A 和一个单位矩阵 E 组合成大矩阵 (A, E)，用初等行变换将大矩阵中的 A 变为 E，则会得到 (E, A^{-1}) 的形式。

复杂度	$O(n^3)$	
输入	A	原矩阵
	C	逆矩阵
	n	矩阵的阶数

```
typedef const vector<double>& Vecref;
typedef       vector<double>  Vecdbf;
inline vector<double> operator*(Vecref a, double b)
{
    int n = a.size(); vector<double> res(n, 0);
    for (int i = 0; i < n; i++) res[i] = a[i] * b;
    return res;
}
inline vector<double> operator-(Vecref a, Vecref b)
{
    int n = a.size(); vector<double> res(n, 0);
    for (int i = 0; i < n; i++) res[i] = a[i] - b[i];
    return res;
}

inline void inverse(Vecdbf A[], Vecdbf C[], int n)
{
    for (int i = 0; i < n; i++) C[i] = Vecdbf(n, 0);
    for (int i = 0; i < n; i++) C[i][i] = 1;
    for (int i = 0; i < n; i++)
    {
        for (int j = i; j < n; j++) if (fabs(A[j][i]) > 0)
        {
            swap(A[i], A[j]);
```

```
25              swap(C[i], C[j]);
26              break;
27          }
28          C[i] = C[i] * (1.0 / A[i][i]);
29          A[i] = A[i] * (1.0 / A[i][i]);
30          for (int j = 0; j < n; j++) if (j != i && fabs(A[j][i]) > 0)
31          {
32              C[j] = C[j] - C[i] * A[j][i];
33              A[j] = A[j] - A[i] * A[j][i];
34          }
35      }
36  }
```

2.4 线性递推

已知 $f_x = a_{x-1} \times f_{x-1} + a_{x-2} \times f_{x-2} + ... + a_{x-n} \times f_{x-n}$ 和 $f_0, f_1, ..., f_{n-1}$，对给定的 t，求 f_t。
将这个递推式看做一个矩阵与一个列向量的乘积的形式。

$$A = \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ a_{n-1} & a_{n-2} & a_{n-3} & \cdots & a_0 \end{pmatrix}, B = \begin{pmatrix} f_{x-n} \\ f_{x-n+1} \\ \vdots \\ f_{x-2} \\ f_{x-1} \end{pmatrix}$$

所以可以用快速幂加速。

复杂度	$O(n^3 logt)$	
输入	a	常系数数组
	b	初值数组
	n	数组大小
	t	求解的项数
输出		f_t

```
1  // f_x = a_x-1 * f_x-1 + a_x-2 * f_x-2 + ... + a_x-n * f_x-n
2  // More specific, fn = a_n-1 * f_n-1 + a_n-2 * f_n-2 + ... + a_0 * f_0
3  // Create A[n][n] and B[n] to calculate f(t)
4  // eg. f3 = 2 * f2 + 3 * f1 - f0, f0 = 1, f1 = 2, f2 = 4.
5  // => a[] = {-1, 3, 2}, b[] = {1, 2, 4}.
6  int solve(int a[], int b[], int n, int t)
7  {
8      Matrix M, F, E;
9      M.clear(); M.n = M.m = n;
10     F.clear(); F.n = n; F.m = 1;
11     E.clear(); E.n = E.m = n;
12     for (int i = 1; i < n; i++) M.a[i - 1][i] = 1;
13     for (int i = 0; i < n; i++)
14     {
15         M.a[n - 1][i] = a[i];
16         F.a[i][0] = b[i];
17         E.a[i][i] = 1;
18     }
19     if (t < n) return F.a[t][0];
20     while (t)
21     {
22         if (t & 1) { E = E * M; t--; }
23         else { t /= 2; M = M * M; }
24     }F = E * F;
25     return F.a[0][0];
26 }
```

2.5 矩阵快速幂

模板题，针对一类转移方程的参数较小，可以利用快速幂加速的一类问题。故给出一个完整的题目模板，使用时务必对自己的系数矩阵，初值矩阵做到清楚清晰，否则会出错。

现给出例题与模板，HDU 4686

$$
\begin{aligned}
1 &= 1 \\
a_i &= AX \times a_{i-1} + AY \times 1 \\
b_i &= BX \times b_{i-1} + BY \times 1 \\
a_i b_i &= (AX \times a_{i-1} + AY \times 1) + (BX \times b_{i-1} + BY \times 1) \\
&= AX \times BX \times a_{i-1} b_{i-1} + AY \times BY + AX \times BY \times a_{i-1} + AY \times BX \times b_{i-1} \\
AoD(i) &= AoD(i-1) + a_{i-1} b_{i-1}
\end{aligned}
$$

$$
\begin{pmatrix} 1 \\ a_n \\ b_n \\ a_n b_n \\ AoD(n) \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ AY & AX & 0 & 0 & 0 \\ BY & 0 & BX & 0 & 0 \\ AY \times BY & AX \times BY & AY \times BX & AX \times BX & 0 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix}^n \begin{pmatrix} 1 \\ a_0 \\ b_0 \\ a_0 b_0 \\ 0 \end{pmatrix}
$$

```cpp
/*******************************************************************************
*                      ----Stay Hungry Stay Foolish----                        *
*    @author    :   Shen                                                      *
*    @name      :   HDU 4686                                                  *
*******************************************************************************/

#include <iostream>
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;
typedef long long int64;

const int MaxN = 5;
const int MaxM = 5;
const int Mod = 1000000007;

struct Matrix
{
    int n, m;
    int64 mat[MaxN][MaxM];
    Matrix(): n(-1), m(-1){}
    Matrix(int _n, int _m): n(_n), m(_m)
    {
        memset(mat, 0, sizeof(mat));
    }
    void Unit(int _s)
    {
        n = _s; m = _s;
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                mat[i][j] = (i == j)? 1: 0;
    }
    void print()
    {
        printf("n = %d, m =  %d\n", n, m);
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < m; j++)
                printf("%8d", mat[i][j]);
            printf("\n");
        }
    }
};

Matrix add_mod(const Matrix& a, const Matrix& b, const int64 mod)
```

```
{
    Matrix ans(a.n, a.m);
    for (int i = 0; i < a.n; i++) for (int j = 0; j < a.m; j++)
        ans.mat[i][j] = (a.mat[i][j] + b.mat[i][j]) % mod;
    return ans;
}

Matrix mul(const Matrix& a, const Matrix& b)
{
    Matrix ans(a.n, b.m);
    for (int i = 0; i < a.n; i++) for (int j = 0; j < b.m; j++)
    {
        int64 tmp = 0;
        for (int k = 0; k < a.m; k++)
            tmp += a.mat[i][k] * b.mat[k][j];
        ans.mat[i][j] = tmp;
    }
    return ans;
}

Matrix mul_mod(const Matrix& a, const Matrix& b, const int mod)
{
    Matrix ans(a.n, b.m);
    for (int i = 0; i < a.n; i++) for (int j = 0; j < b.m; j++)
    {
        int64 tmp = 0;
        for (int k = 0; k < a.m; k++)
            tmp += (a.mat[i][k] * b.mat[k][j]) % mod;
        ans.mat[i][j] = tmp % mod;
    }
    return ans;
}

Matrix pow_mod(const Matrix& a, int64 k, const int mod)
{
    Matrix p(a.n,a.m), ans(a.n,a.m);
    p = a; ans = a;
    ans.Unit(a.n);
    if (k == 0) return ans;
    else if (k == 1) return a;
    else
    {
        while (k)
        {
            if (k & 1) { ans=mul_mod(ans, p, mod); k--; }
            else { k /= 2; p = mul_mod(p, p, mod); }
        }
        return ans;
    }
}

int64 n;
int64 a0, ax, ay;
int64 b0, bx, by;

void solve()
{
    Matrix ans(5, 1);

    Matrix beg(5, 1);
    beg.mat[0][0] = 1;
    beg.mat[1][0] = a0;
    beg.mat[2][0] = b0;
    beg.mat[3][0] = a0 * b0 % Mod;
    beg.mat[4][0] = 0;

    Matrix cef(5, 5);
    memset(cef.mat, 0, sizeof(cef.mat));
```

```
115     cef.mat[0][0] = 1;
116     cef.mat[1][0] = ay % Mod; cef.mat[1][1] = ax % Mod;
117     cef.mat[2][0] = by % Mod; cef.mat[2][2] = bx % Mod;
118     cef.mat[3][0] = ay * by % Mod; cef.mat[3][1] = ax * by % Mod;
119     cef.mat[3][2] = ay * bx % Mod; cef.mat[3][3] = ax * bx % Mod;
120     cef.mat[4][3] = 1; cef.mat[4][4] = 1;
121
122     Matrix tmp(5, 5);
123     tmp = pow_mod(cef, n, Mod);
124     ans = mul_mod(tmp, beg, Mod);
125
126     cout << ans.mat[4][0] << endl;
127     return;
128 }
129
130 int main()
131 {
132     while (cin >> n)
133     {
134         cin >> a0 >> ax >> ay;
135         cin >> b0 >> bx >> by;
136         solve();
137     }
138     return 0;
139 }
```

3 整除与剩余

3.1 欧几里得算法

3.1.1 辗转相除法

求两个数 a, b 的最大公约数 gcd(a, b)。

根据 $gcd(a,b) = gcd(b, a-b)$ 可以进一步推导出 $gcd(a,b) = gcd(b, a\%b)$ ，所以只需要不停地迭代即可计算出结果。但是这个并不能有效地处理负数问题。所以在 Miller - Robin 测试中，使用的是另一个非递归版的辗转相除法版本。

复杂度	$O(\log N)$	其中 N 与 a, b 同阶
输入	a, b	两个整数
输出		a, b 的最大公约数

```
1 int64 gcd(int64 a, int64 b) { return b == 0? a: gcd(b, a % b); }
```

3.1.2 最小公倍数

求两个数 a, b 的最小公倍数 lcm(a, b)。

根据 $gcd(a,b) \times lcm(a,b) = a \times b$ 可以计算出两个数的最小公倍数。唯一要注意的是，直接先乘再除可能会在乘法部分发生精度溢出，所以必须先除最大公约数，再乘。

复杂度	$O(\log N)$	其中 N 与 a, b 同阶
输入	a, b	两个整数
输出		a, b 的最小公倍数

```
1 int64 lcm(int64 a, int64 b) { return a / gcd(a, b) * b; }
```

3.1.3 拓展欧几里得

求出 a, b 的最大公约数，并且同时计算出一组可能的 x, y 使得 $ax + by = (a, b)$

复杂度	$O(\log N)$	其中 N 与 a, b 同阶
输入	a, b	两个整数
	x, y	引用，用于返回一组解
输出		a, b 的最大公约数

```
1 int64 gcd_ex(int64 a, int64 b, int64& x, int64& y)
2 {
3     if (b == 0) { x = 1; y = 0; return a; }
4     int64 d = gcd_ex(b, a % b, y, x);
5     y = y - a / b * x;
6     return d;
7 }
```

3.2 整数快速幂

3.2.1 整数模乘法

用于计算 $a \times b \ mod \ m$

复杂度	$O(\log N)$	其中 N 与 a, b 同阶
输入	a, b	两个整数
	m	模数
输出		$a \times b \ mod \ m$

```
int64 mul_mod(int64 a, int64 b, int64 m)
{
    int64 t = 0; a %= m; b %= m;
    while (b)
    {
        if (b & 1) t += a, t %= m;
        a <<= 1; a %= m; b >>= 1;
    }
    return t;
}
```

3.2.2 整数快速幂

用于计算 $a^b \ mod \ m$

复杂度	$O(\log N)$	其中 N 与 b 同阶
输入	a, b	两个整数
	m	模数
输出		$a^b \ mod \ m$

```
int64 pow_mod(int64 a, int64 b, int64 m)
{
    int64 ans = 1; a %= m;
    while (b)
    {
        if (b & 1) { ans = mul_mod(ans, a, m); b--; }
        b >>= 1; a = mul_mod(a, a, m);
    }
    return ans;
}
```

3.3 一元一次模线性方程

3.3.1 求特解

用于计算 $ax \equiv b(mod \ m)$ 的一个特解，如果没有解则返回 m 值本身

复杂度	$O(\log N)$	其中 N 与 a, b 同阶
输入	a, b	两个整数
	m	模数
输出		一个特解 x

```
// ax = b (mod m)
// returning m when no solusion
int64 solve(int64 a, int64 b, int64 m)
{
    int64 x, y; int64 d = gcd_ex(a, m, x, y);
    if (b % d == 0)
    {
        x %= m; while (x < 0) x += m; x %= m;
        return x * (b / d) % (m / d);
    }
    else return m;
}
```

3.3.2 求区间全体解

用于计算 $ax \equiv b(mod \ m)$ 在区间 $[0, m)$ 的所有解

复杂度	$O(\log N)$	其中 N 与 a, b 同阶

输入	a, b	两个整数
	m	模数
输出	vector<int64>	所有解

```
// ax = b (mod m), all x in [0, m)
// returning empty vector<int64> when no solusion
vector<int64> solve(int64 a, int64 b, int64 m)
{
    int64 x, y; int64 d = gcd_ex(a, m, x, y);
    vector<int64> ans; ans.clear();
    if (b % d == 0)
    {
        x %= m; while (x < 0) x += m; x %= m;
        ans.push_back(x * (b / d) % (m / d));
        for (int64 i = 1; i < d; i++)
            ans.push_back((ans[0] + i * m / d) % m);
    }
    return ans;
}
```

3.4 中国剩余定理

3.4.1 中国剩余定理

用于计算方程组 $x \equiv a_i(mod\ m_i)$ 的一个特解，其中要求所有的模数两两互素。
简要的求解过程如下：
1. $M_0 = m_1 \times m_2 \times ...m_n$
2. c_i 是方程 $M_i x \equiv 1(mod\ m_i)$ 的一个特解，其中，$M_i = M_0/m_i$。
3. $x \equiv a_1c_1M_1 + a_2c_2M_2 + ... + a_nc_nM_n(mod\ M_0)$

复杂度	$O(nlogM)$	其中 M 与每一个 m_i 同阶
输入	a[]	方程的常数
	m[]	每一个方程的模数
	n	方程的个数
输出		一个特解

```
// x = ai (mod mi), for i := [0, n)
// @return result;
int64 CRT(int64 a[],int64 m[],int64 n)
{
    int64 M = 1, res = 0;
    for (int i = 0; i < n; i++) M *= m[i];
    for (int i = 0; i < n; i++)
    {
        int64 x, y, tm = M / m[i];
        gcd_ex(tm, m[i], x, y);
        res = (res + tm * x * a[i]) % M;
    }
    return (res + M) % M;
}
```

3.4.2 拓展中国剩余定理

用于计算方程组 $x \equiv a_i(mod\ m_i)$ 的一个特解，其中不必要求所有的模数两两互素。
此时用的方法是方程两两合并的方法。

复杂度	$O(nlogM)$	其中 M 与每一个 m_i 同阶
输入	a[]	方程的常数
	m[]	每一个方程的模数
	n	方程的个数

输出		一个特解

```
// More effective mod function, returning a positive num
inline int64 mod(int64 a, int64 m) { return a % m + (a % m > 0? 0: m); }

// x = ai (mod mi), for i := [0, n)
// @return legal Equalion? result: -1;
int64 CRT_ex(int n, int a[], int m[])
{
    if (n == 1 && a[0] == 0) return m[0];
    int64 ans = a[0], lcm = m[0];
    bool flag = true;
    for (int i = 1; i < n; i++)
    {
        int64 x, y, gcd;
        gcd = gcd_ex(lcm, m[i], x, y);
        if ((a[i] - ans) % gcd) { flag = false; break; }
        int64 tmp = lcm * mod((a[i] - ans) / gcd * x, m[i] / gcd);
        lcm = lcm / gcd * m[i];
        ans = mod(ans + tmp, lcm);
    }
    return flag? ans: -1;
}
```

3.5 运算推广

3.5.1 乘法逆元

用于计算元素在模数为 m 时的乘法逆元（如果存在），即 $ax \equiv 1(mod\ m)$ 的一个特解。有两种计算方法，一种是利用扩展欧几里得直接计算，另一种是利用欧拉函数的性质去进行计算。两者的时间复杂度近似相同，返回结果一样。

复杂度	$O(\log N)$	其中 N 与 a, mod 同阶
输入	a	所要计算逆元的整数
	m	模数
输出		乘法逆元 a^{-1}

```
// ax = 1 (mod m)
int64 inv(int64 a, int64 mod)
{
    int64 x, y;
    int64 t = gcd_ex(a, mod, x, y);
    return (x % mod + mod) % mod;
}
// using eular function
int64 inv(int64 a, int64 mod)
{
    return pow_mod(a, mod - 2, mod);
}
```

3.5.2 求原根

用于求一个在模数为 m 的意义下的原根，其中 m 为素数。

定义，若 $m > 1$，$(a, m) = 1$，则使得同余式 $a^{\gamma} \equiv 1(mod\ m)$ 成立的最小正整数 γ 叫做 a 对模 m 的指数，记做 $\gamma = Ord_m(a)$。

若 $\gamma = Ord_m(a) = \varphi(m)$，即 $a^{\varphi(m)} \equiv 1(mod\ m)$，此时称 a 为在模数为 m 的意义下的原根。

原根的分布比较广，并且最小的原根通常也较小，故可以通过从小到大枚举正整数来寻找一个原根。对于一个待检查的 m，对 $m-1$ 的每一个素因子 a，检查 $g^{(p-1)/a} \equiv 1(mod\ m)$ 是否成立，如果成立则说明 g 不是原根。

代码中，为了明确指出 m 必须为素数，故统一用 p 来表示。

复杂度	$O(\sqrt{N}\log N)$	其中 N 与 p 同阶
输入	p	模数
输出		p 的原根

```
// a ^ phi(p) = 1 (mod p)
vector<int64> a;

bool g_test(int64 g, int64 p)
{
    for (int64 i = 0; i < a.size(); i++)
        if (pow_mod(g, (p - 1LL) / a[i], p) == 1LL)
            return 0;
    return 1;
}

int64 primitive_root(int64 p)
{
    int64 tmp = p - 1;
    for (int64 i = 2; i <= tmp / i; i++) if (tmp % i == 0)
    {
        a.push_back(i);
        while (tmp % i == 0) tmp /= i;
    }
    if (tmp != 1) a.push_back(tmp);
    int64 g = 1;
    while (1) { if (g_test(g, p)) return g; g++; }
}
```

3.5.3 勒让德符号

用于求 d 对 p 的勒让德符号，$\frac{d}{p}$

其定义如下，

$$(\frac{d}{p}) = \begin{cases} 1 & \text{d 是模 p 的平方剩余} \\ -1 & \text{d 是模 p 的平方非剩余} \\ 0 & \text{d 整除 p} \end{cases}$$

勒让德符号可以直接用欧拉判别条件进行计算，即 $(\frac{d}{p}) \equiv a^{\frac{p-1}{2}} (mod\ p)$ 。

复杂度	$O(\log N)$	其中 N 与 p 同阶
输入	a	所要计算的整数
	p	模数
输出		$(\frac{d}{p})$

```
// (a|p) =
// 1, when x ^ 2 = a (mod p) has solusion
// -1, when x ^ 2 = a (mod p) has no solusion
// 0, when p|a
int64 Legendre(int64 d, int64 p)
{
    int64 coef = (d > 0)? 1: (((p - 1) % 4 == 0)? 1: -1);
    d = (d > 0)? d: -d;
    d %= p;
    if (pow_mod(d, (p - 1) / 2, p) == 1) return coef;
    else return -coef;
}
```

3.5.4 平方剩余

用于求方程 $x^2 \equiv a(mod\ m)$ 的最小整数解。

先判断是否有解，然后根据剩余类进行特殊判断。

复杂度	$O(log^2N)$	其中 N 与 m 同阶
输入	a	方程系数
	m	模数
输出		平方剩余 x

```
// x ^ 2 = a (mod m), solve x
int64 modsqr(int64 a, int64 m)
{
    int64 b, k, i, x; a %= m;
    if (m == 2) return a % m;
    if (pow_mod(a, (m - 1) / 2, m) == 1)
    {
        if (m % 4 != 3)
        {
            for (b = 1; pow_mod(b, (m - 1) / 2, m) == 1; b++);
            i = (m - 1) / 2; k = 0;
            while (true)
            {
                i /= 2; k /= 2;
                int64 h1 = pow_mod(a, i, m), h2 = pow_mod(b, k, m);
                if ((h1 * h2 + 1) % m == 0) k += (m - 1) / 2;
                if (i % 2 != 0) break;
            }
            int64 t1 = pow_mod(a, (i + 1) / 2, m);
            int64 t2 = pow_mod(b, k / 2, m);
            x = mul_mod(t1, t2, m);
        }
        else x = pow_mod(a, (m + 1) / 4, m);
        if (x * 2 > m) x = m - x;
        return x;
    }
    else return -1;
}
```

3.5.5 离散对数

用于求方程 $x^y \equiv n(mod\ m)$ 的最小整数解 y，其中 p 为素数。若无解则返回 -1。

使用 giant-step baby-step 算法。令 $s = \lfloor\sqrt{m}\rfloor$，则有 $y = b \times s + r(0 \le r < s)$，即有 $x^y = x^{b\times s} \times x^r$。将所有的 x^r 放入有序表中，从小到大枚举 b，得到：$x^{b\times s} \times x^r = n$。

把 x^r 看成未知数解模线性方程。若解 x^r 能够在有序表中二分查找得到，则停止枚举，此时，$y = b \times s + r$。

复杂度	$O(\sqrt{m})$	
输入	x, n	方程系数
	m	模数，且为素数
输出		离散对数 y

```
// x ^ y = n (mod m), solve y
int64 discrete_log(int64 x, int64 n, int64 m)
{
    map<int64, int> rec;
    int s = (int)(sqrt((double)m));
    for (; (int64)s * s <= m; ) s++;
    int64 cur = 1;
    for (int i = 0; i < s; i++)
    {
        rec[cur] = i;
        cur = cur * x % m;
    }
    int64 mul = cur; cur = 1;
```

```
    for (int i = 0; i < s; i++)
    {
        int64 more = (int64)n * pow_mod(cur, m - 2, m) % m;
        if (rec.count(more)) return i * s + rec[more];
        cur = cur * mul % m;
    }
    return -1;
}
```

3.5.6 N 次剩余

用于求方程 $x^N \equiv a(mod\ p)$ 的所有整数解 y，其中 p 为素数。若无解则返回一个空 vector<int>。

令 g 为 p 的原根，因为 p 为素数，则有 $\varphi(p) = p-1$，所以找到原根 g 就可以将 $\{1, 2, ..., p-1\}$ 的数与 $\{g^1, g^2, ..., g^{p-1}\}$ 建立一一对应关系。

令 $x = g^y, a = g^t$，则有：

$g^{y \times N} \equiv g^t(mod\ p)$

又由于 p 是素数，所以方程的左右都不可以为 0。这样就可以将这 $p-1$ 个取值与指数建立对应关系。此时问题被转化为：

$y \times N \equiv t(mod\ (p-1))$

对 y 解模线性方程即可。而 $a = g^t$ 可以用离散对数来求解。

复杂度	$O(\sqrt{p})$	
输入	a, N	方程系数
	p	模数，且为素数
输出	vector<int>	全体整数解 $\{x\}$

```
// x ^ N = a (mod p), solve x
vector<int64> residue(int64 N, int64 a, int64 p)
{
    int64 g = primitive_root(p);
    int64 m = discrete_log(g, a, p);
    vector<int64> ret;
    if (a == 0) { ret.push_back(0); return ret; }
    if (m == -1) return ret;
    int64 A = N, B = p - 1, C = m, x, y;
    int64 d = gcd_ex(A, B, x, y);
    if (C % d != 0) return ret;
    x = x * (C / d) % B;
    int64 delta = B / d;
    for (int i = 0; i < d; i++)
    {
        x = ((x + delta) % B + B) % B;
        ret.push_back((pow_mod(g, x, p)));
    }
    sort(ret.begin(), ret.end());
    ret.erase(unique(ret.begin(), ret.end()), ret.end());
    return ret;
}
```

3.6 组合数求模

提前说明，以下三种方法各有优缺点，请按需使用！

3.6.1 朴素递推

顾名思义，只要把除法改为乘上逆元就行了。所以朴素递推法更适合数据量较小的 (MaxM$\leq 10^6$) 组合数求模，如果数据量太大，预处理逆元可以分解到每一次的查询中。而且此方法很容易 MLE。

复杂度	$O(M)$ - $O(1)$	预处理 - 查询
输入	n, k	参数
输出		组合数 C_n^k

```
const int MaxM = 100005;
const int64 MOD = 1000000009;
int64 inv[MaxM]; // inv, a * inv(a) % p = 1
int64 fac[MaxM]; // fact, 1 * 2 * 3 * ...
int64 rfc[MaxM]; // inv-fact, inv(1) * inv(2) * inv(3) * ...

void init()
{
    inv[0] = inv[1] = 1;
    fac[0] = fac[1] = 1;
    rfc[0] = rfc[1] = 1;
    for (int i = 2; i < MaxM; i++)
    {
        inv[i] = ((MOD - MOD / i) * inv[MOD % i]) % MOD;
        fac[i] = (fac[i - 1] * i) % MOD;
        rfc[i] = (rfc[i - 1] * inv[i]) % MOD;
    }
}

inline int64 c(int64 n, int64 k)
{
    return (fac[n] * rfc[k] % MOD) * rfc[n - k] % MOD;
}
```

3.6.2 逆元求解

此方法只需体现预处理出所有的阶乘，然后每一次查询时再去计算逆元。适用于 M 值适中 (MaxM≤ 2×10^6)，且对 MOD 无数据大小要求的组合数求模运算。

复杂度	$O(M)$ - $O(\log MOD)$	预处理 - 查询
输入	n, k	参数
输出		组合数 C_n^k

```
const int MaxM = 2000005;
const int64 MOD = 1000000009;
int64 fac[MaxM]; // fact, 1 * 2 * 3 * ...
void init()
{
    fac[0] = fac[1] = 1;
    for (int i = 2; i < MaxM; i++)
        fac[i] = (fac[i - 1] * i) % MOD;
}

int64 c(int64 n, int64 k)
{
    int64 x = 0, y = 0;
    int64 tmp = mul_mod(fac[k], fac[n - k], MOD);
    gcd_ex(tmp, MOD, x, y);
    x = (x % MOD + MOD) % MOD;
    return mul_mod(fac[n], x, MOD);
}
```

3.6.3 Lucas 算法

Lucas 算法的核心思路是将目标用 p 进制来表示，这样就可以同时实现加速分解与计算，十分快捷与方便。该算法对 n, k 的数据规模没有限制，缺点是只能够计算模数较小的组合数，即小素数的组合数求模。一般而言，模数 MOD 的数量级不超过 10^6。

由于核心算法是转写成 p 进制的递归分解，其查询时的时间复杂度几乎介于对数复杂度与常数复杂度之间。

复杂度	$O(MOD)$ - $O(k)$	预处理 - 查询
输入	n, k	参数
输出		组合数 C_n^k

```
const int64 MOD = 10007;
int64 fac[MOD]; // fact, 1 * 2 * 3 * ...
void init()
{
    fac[0] = fac[1] = 1;
    for (int i = 2; i < MaxM; i++)
        fac[i] = (fac[i - 1] * i) % MOD;
}

int64 c(int64 n, int64 k)
{
    int64 x = 0, y = 0;
    int64 tmp = mul_mod(fac[k], fac[n - k], MOD);
    gcd_ex(tmp, MOD, x, y);
    x = (x % MOD + MOD) % MOD;
    return mul_mod(fac[n], x, MOD);
}

int64 Lucas(int64 n, int64 k)
{
    if (k == 0) return 1;
    int64 t1 = c(n % MOD, k % MOD);
    int64 t2 = Lucas(n / MOD, k / MOD);
    return t1 * t2 % MOD;
}
```

4 素性与函数

4.1 素数筛

4.1.1 埃氏筛

最普通的素数筛，将素数的倍数全部筛去从而得到素数表。

复杂度	$O(N \log N)$	
全局	isPrime[]	是否为素数
	prime[]	素数表，从下标 0 开始
	tot	素数个数

```
const int MaxN = 1000005;
bool isPrime[MaxN];
int tot, prime[MaxN];

void getPrime()
{
    fill(isPrime, isPrime + MaxN, true);
    isPrime[0] = isPrime[1] = 0; tot = 0;
    for (int i = 2; i < MaxN; i++) if (isPrime[i])
    {
        if (n / i < i) break;
        for (int j = i * i; j < MaxN; j += i)
            isPrime[j] = false;
    }
    for (int i = 2; i < MaxN; i++) if (isPrime)
        prime[tot++] = i;
}
```

4.1.2 线性筛

近似线性时间复杂度的素数筛。核心是每次筛数只筛到本身与当前的最大素数为止。相当于将筛的任务均摊下去，避免了重复筛的情况。

复杂度	$O(N)$	
全局	isPrime[]	是否为素数
	prime[]	素数表，从下标 0 开始
	tot	素数个数

```
const int MaxN = 1000005;
bool isPrime[MaxN];
int tot, prime[MaxN];

void getPrime()
{
    fill(isPrime, isPrime + MaxN, true);
    isPrime[0] = isPrime[1] = 0; tot = 0;
    for (int i = 2; i < MaxN; i++)
    {
        if (isPrime[i]) prime[tot++] = i;
        for (int j = 0; j < tot; j++)
        {
            if (i * prime[j] >= MaxN) break;
            isPrime[i * prime[j]] = false;
            if (i % prime[j] == 0) break;
        }
    }
}
```

4.1.3 区间筛

用于求区间 $[L,R]$ 中的所有素数。先用线性筛预处理出一部分，近似是这个区间大小 $[1,2^{15}]$，然后在用素数去筛所求区间，从而得到素数表。

复杂度	$O(N)$	
预处理	isPrime[]	是否为素数
	prime[]	素数表，从下标 0 开始
	tot	素数个数
复杂度	$O(R-L)$	
区间筛	notPrime[]	是否不为素数，与 isPrime 正好相反
	prime2[]	区间的素数表，从下标 0 开始
	tot2	区间的素数个数

```
const int MaxN = 1000005;
bool isPrime[MaxN];
int tot, prime[MaxN];

void getPrime()
{
    fill(isPrime, isPrime + MaxN, true);
    isPrime[0] = isPrime[1] = 0; tot = 0;
    for (int i = 2; i < MaxN; i++)
    {
        if (isPrime[i]) prime[tot++] = i;
        for (int j = 0; j < tot; j++)
        {
            if (i * prime[j] >= MaxN) break;
            isPrime[i * prime[j]] = false;
            if (i % prime[j] == 0) break;
        }
    }
}

bool notPrime[MaxN];
int cnt, prime2[MaxN];
void getPrime2(int L, int R)
{
    fill(notPrime, notPrime + MaxN, false);
    if (L < 2) L = 2;
    for (int i = 0; i <= tot && (int64)prime[i] * prime[i] <= R; i++)
    {
        int s = L / prime[i] + (L % prime[i] > 0);
        if (s == 1) s = 2;
        for (int j = s; (int64)j * prime[i] <= R; j++)
            if ((int64)j * prime[i] >= L)
                notPrime[j * prime[i] - L] = true;
    }
    cnt = 0;
    for (int i = 0; i <= R - L; i++) if (!notPrime[i])
        prime2[cnt++] = i + L;
}
```

4.2 Miller-Robin 判别法

用于快速判断一个数是否为素数。具体做法是通过反复的欧拉定理与二次剩余特判来处理。如果多次判断之后仍然是真值，则说明该数有很大可能为素数。

由于是随机性算法，不能完完全全确保正确性。正确概率在 99.9%，适当增加测试次数可以提高正确性。

复杂度	$O(k\log N)$	
全局	25	执行 25 次随机判断
输入	n	需要判断的整数
输出	bool	若为素数返回真，反之返回否

```cpp
// <!--encoding UTF-8 UTF编码-8--!>
/*************************************************************************
*                      ----Stay Hungry Stay Foolish----                  *
*    @author    :   Shen                                                 *
*    @name      :   poj 1811                                             *
*************************************************************************/

#include <ctime>
#include <cstdio>
#include <cstring>
#include <iostream>
#include <algorithm>
using namespace std;
typedef long long int64;
template<class T>inline bool updateMin(T& a, T b){ return a > b ? a = b, 1: 0; }
template<class T>inline bool updateMax(T& a, T b){ return a < b ? a = b, 1: 0; }
inline int    nextInt() { int x; scanf("%d", &x); return x; }
inline int64  nextI64() { int64  d; cin >> d; return d; }
inline char   nextChr() { scanf(" "); return getchar(); }
inline double nextDbf() { double x; scanf("%lf", &x); return x; }
inline int64  next64d() { int64 d; scanf("%I64d",&d); return d; }

const int64 MaxN = 105;
int64 gcd(int64 a, int64 b)
{
    if (a == 0) return 1;
    if (a < 0) return gcd(-a, b);
    while (b)
    {
        int64 t = a; a = b; b = t % b;
    }
    return a;

}

int64 mul_mod(int64 a, int64 b, int64 m)
{
    int64 t = 0; a %= m; b %= m;
    while (b)
    {
        if (b & 1) t += a, t = (t >= m)? t - m: t;
        a <<= 1; a = (a >= m)? a - m: a; b >>= 1;
    }
    return t;
}

int64 pow_mod(int64 a, int64 b, int64 m)
{
    int64 ans = 1; a %= m;
    while (b)
    {
        if (b & 1) ans = mul_mod(ans, a, m);
        b >>= 1; a = mul_mod(a, a, m);
    }
    return ans;
}

bool test(int64 a, int64 n, int64 x, int64 t)
{
    int64 ret = pow_mod(a, x, n);
    int64 last = ret;
    for (int i = 1; i <= t; i++)
    {
        ret = mul_mod(ret, ret, n);
        if (ret == 1 && last != 1 && last != n - 1)
            return true;
        last = ret;
    }
```

```
    if (ret != 1) return true;
    else return false;
}

bool isPrime(int64 n)
{
    int64 x = n - 1, t = 0;
    while ((x & 1) == 0) { x >>= 1; t++; }
    bool flag = 1;
    if (t >= 1 && (x & 1) == 1)
    {
        for (int k = 0; k < 25; k++)
        {
            int64 a = rand() % (n - 1) + 1;
            if (test(a, n, x, t)) { flag = 1; break; }
            flag = 0;
        }
    }
    if (!flag || n == 2) return 1;
    return 0;
}

int64 Pollard_rho(int64 x, int64 c)
{
    int64 i = 1, k = 2;
    int64 x0 = rand() % (x - 1) + 1;
    int64 y = x0;
    while (true)
    {
        i++;
        x0 = (mul_mod(x0, x0, x) + c) % x;
        int64 d = gcd(y - x0, x);
        if (d != 1 && d != x) return d;
        if (y == x0) return x;
        if (i == k) { y = x0; k += k; }
    }
}

int64 tot, result[MaxN];
void findfac(int64 n)
{
    if (n == 1) return;
    if (isPrime(n)) { result[tot++] = n; return; }
    int64 p = n;
    while (p >= n) p = Pollard_rho(p, rand() % (n - 1) + 1);
    findfac(p); findfac(n / p);
}

int t; int64 n;

void solve()
{
    n = next64d();
    if (isPrime(n)) puts("Prime");
    else
    {
        tot = 0; findfac(n);
        int64 ans = result[0];
        for (int i = 0; i < tot; i++)
            updateMin(ans, result[i]);
        printf("%I64d\n", ans);
    }
}

int main()
{
    srand(time(0));
    t = nextInt(); while (t--) solve();
```

```
137     return 0;
138 }
```

4.3 素因数分解

4.3.1 朴素分解

朴素分解，特判最后的剩余项是否为 1，若为 1 则说明已经除尽，反之说明剩余项也为其素因子。

复杂度	$O(\sqrt{N})$	
输入	n	需要分解的整数
	a[]	存储素因子
	b[]	存储素因子的次数
	tot	存储素因子的个数

```
1  const int MaxN = 1005;
2  void factor(int n, int a[MaxN], int b[MaxN], int& tot)
3  {
4      int now = n; tot = 0;
5      for (int i = 2; i <= n / i; i++) if (now % i == 0)
6      {
7          b[tot] = 0;
8          while (now % i == 0) { ++b[tot]; now /= i; }
9          a[tot++] = i;
10     }
11     if (now != 1) { b[tot] = 1; a[tot++] = now; }
12 }
```

4.3.2 Pollard-rho 方法

用 Pollard-rho 方法实现素因数分解。分解的顺序是随机的，有两种储存方式，数组储存然后排序得到有序的序列，或者直接用 map 储存均可。

给出两个例子，POJ 1811 使用的是数组存储，POJ 2429 使用的是 map 存储。

复杂度	$O(\log N)$	
输入	n	需要分解的整数
1.	tot	存储素因子的个数
	result[]	存储素因子，无顺序
2.	map<int64, int64> result	存储素因子以及次数
	first	value
	second	times

```
1  // <!--encoding UTF-8 UTF编码-8--!>
2  /*************************************************************************
3  *                      ----Stay Hungry Stay Foolish----                  *
4  *    @author    :   Shen                                                 *
5  *    @name      :   poj 2429                                             *
6  *************************************************************************/
7
8  #include <map>
9  #include <ctime>
10 #include <climits>
11 #include <cstdio>
12 #include <cstring>
13 #include <iostream>
14 #include <algorithm>
15 using namespace std;
16 typedef long long int64;
17 template<class T>inline bool updateMin(T& a, T b){ return a > b ? a = b, 1: 0; }
18 template<class T>inline bool updateMax(T& a, T b){ return a < b ? a = b, 1: 0; }
```

```
19 inline int    nextInt() { int x; scanf("%d", &x); return x; }
20 inline int64  nextI64() { int64  d; cin >> d; return d; }
21 inline char   nextChr() { scanf(" "); return getchar(); }
22 inline double nextDbf() { double x; scanf("%lf", &x); return x; }
23 inline int64  next64d() { int64 d; scanf("%I64d",&d); return d; }
24
25 typedef map<int64, int64>::iterator itr;
26
27 const int64 MaxN = 1005;
28 const int64 INF  = ~0ull >> 1;
29 int64 gcd(int64 a, int64 b)
30 {
31     if (a == 0) return 1;
32     if (a < 0) return gcd(-a, b);
33     while (b)
34     {
35         int64 t = a; a = b; b = t % b;
36     }
37     return a;
38
39 }
40
41 int64 mul_mod(int64 a, int64 b, int64 m)
42 {
43     int64 t = 0; a %= m; b %= m;
44     while (b)
45     {
46         if (b & 1) t += a, t = (t >= m)? t - m: t;
47         a <<= 1; a = (a >= m)? a - m: a; b >>= 1;
48     }
49     return t;
50 }
51
52 int64 pow_mod(int64 a, int64 b, int64 m)
53 {
54     int64 ans = 1; a %= m;
55     while (b)
56     {
57         if (b & 1) ans = mul_mod(ans, a, m);
58         b >>= 1; a = mul_mod(a, a, m);
59     }
60     return ans;
61 }
62
63 bool test(int64 a, int64 n, int64 x, int64 t)
64 {
65     int64 ret = pow_mod(a, x, n);
66     int64 last = ret;
67     for (int i = 1; i <= t; i++)
68     {
69         ret = mul_mod(ret, ret, n);
70         if (ret == 1 && last != 1 && last != n - 1)
71             return true;
72         last = ret;
73     }
74     if (ret != 1) return true;
75     else return false;
76 }
77
78 bool isPrime(int64 n)
79 {
80     int64 x = n - 1, t = 0;
81     while ((x & 1) == 0) { x >>= 1; t++; }
82     bool flag = 1;
83     if (t >= 1 && (x & 1) == 1)
84     {
85         for (int k = 0; k < 25; k++)
86         {
```

```
            int64 a = rand() % (n - 1) + 1;
            if (test(a, n, x, t)) { flag = 1; break; }
            flag = 0;
        }
    }
    if (!flag || n == 2) return 1;
    return 0;
}

int64 Pollard_rho(int64 x, int64 c)
{
    int64 i = 1, k = 2;
    int64 x0 = rand() % (x - 1) + 1;
    int64 y = x0;
    while (true)
    {
        i++;
        x0 = (mul_mod(x0, x0, x) + c) % x;
        int64 d = gcd(y - x0, x);
        if (d != 1 && d != x) return d;
        if (y == x0) return x;
        if (i == k) { y = x0; k += k; }
    }
}

int64 tot;
map<int64, int64> result;
// -> first: value, -> second: times
void findfac(int64 n)
{
    if (n == 1) return;
    if (isPrime(n)) { result[n]++; return; }
    int64 p = n;
    while (p >= n) p = Pollard_rho(p, rand() % (n - 1) + 1);
    findfac(p); findfac(n / p);
}

int64 cnt, data[MaxN];
inline int64 powi(int64 n, int64 k)
{
    int64 ans = 1LL;
    while (k) { ans *= n; k--; }
    return ans;
}

void pre()
{
    cnt = 0;
    for (itr i = result.begin(); i != result.end(); i++)
    {
        pair<int64, int64> tmp = *i;
        data[cnt++] = powi(tmp.first, tmp.second);
    }
}

int64 mins = INF;
int64 g, l, aa, bb, c;

void dfs(int64 a, int64 b, int n)
{
    if (a + b >= mins) return;
    if (n == cnt)
    {
        if (a + b < mins) { mins = a + b; aa = a; bb = b; }
        return;
    }
    dfs(a * data[n], b, n + 1);
    dfs(a, b * data[n], n + 1);
```

```
}

void solve()
{
    result.clear(); tot = cnt = 0;
    mins = INF;
    c = l / g; findfac(c);
    pre(); dfs(g, g, 0);
    if (aa > bb) swap(aa, bb);
    printf("%I64d %I64d\n", aa, bb);
}

int main()
{
    while (~scanf("%I64d%I64d", &g, &l)) solve();
    return 0;
}
```

4.4 欧拉函数

欧拉函数 $\varphi(n)$，表示小于或等于 n 的数中，与 n 互素的数的数目。
欧拉函数求值的方法以及欧拉函数的性质如下所示：
1. $\varphi(1) = 1$
2. 若 n 是素数 p 的 k 次幂，则有 $\varphi(n) = p^k - p^{k-1} = (p-1)p^{k-1}$
3. 若 m, n 互素，则有 $\varphi(mn) = \varphi(m)\varphi(n)$

4.4.1 求单值

直接利用定义求解即可。

复杂度	$O(\sqrt{N})$	预处理
输入	n	需要计算的值
输出		欧拉函数的值 $\varphi(n)$

```
int64 calcPhi(int64 n)
{
    int64 ans = n;
    for (int i = 2; i * i <= n; i++) if (n % i == 0)
    {
        ans -= ans / i;
        while (n % i == 0) n /= i;
    }
    if (n > 1) ans -= ans / n;
    return ans;
}
```

4.4.2 筛法求欧拉函数

根据欧拉函数的定义，可以推导出欧拉函数的递推式。
令 p 为 N 的最小素因数，若 $p^2|N$，$\varphi(N) = \varphi(\frac{N}{p}) \times p$，否则 $\varphi(N) = \varphi(\frac{N}{p}) \times (p-1)$

复杂度	$O(N \log N)$	
全局	phi[]	欧拉函数的值

```
const int MaxN = 3000005;
int phi[MaxN];

void getPhi()
{
    fill(phi, phi + MaxN, 0); phi[1] = 1;
```

```
    for (int i = 2; i < MaxN; i++) if (!phi[i])
        for (int j = i; j < MaxN; j += i)
    {
        if (!phi[j]) phi[j] = j;
        phi[j] = phi[j] / i * (i - 1);
    }
}
```

4.4.3 线性筛求欧拉函数

类似素数的线性筛法，可以将计算欧拉函数的筛法优化至线性时间复杂度。即在筛素数的同时，用递推式的结论，将其计算。

复杂度	$O(N)$	
全局	isPrime[]	是否为素数
	prime[]	素数表，从下标 0 开始
	tot	素数个数
	phi[]	欧拉函数的值

```
const int MaxN = 1000005;
bool isPrime[MaxN];
int tot, prime[MaxN], phi[MaxN];

void getPhi_Prime()
{
    fill(isPrime, isPrime + MaxN, true);
    isPrime[0] = isPrime[1] = 0;
    phi[1] = 1; tot = 0;
    for (int i = 2; i < MaxN; i++)
    {
        if (isPrime[i]) { prime[tot++]; phi[i] = i - 1; }
        for (int j = 0; j < tot; j++)
        {
            if (i * prime[j] >= MaxN) break;
            isPrime[i * prime[j]] = false;
            if (i % prime[j] != 0)
                phi[i * prime[j]] = phi[i] * (prime[j] - 1);
            else { phi[i * prime[j]] = phi[i] * prime[j]; break; }
        }
    }
}
```

4.5 Möbius 函数

Möbius 函数 $\mu(n)$ 是做 Möbius 反演的时候一个很重要的系数。

Möbius 函数的定义如下：如果 i 的素因数分解式内有任意一个大于 1 的指数，则有 $\mu(n) = 0$，否则 $\mu(n) = (-1)^s$，其中 s 是 i 的素因数分解式内素数个数。

定义一数论函数 $[x]$，表示不大于 x 的最大整数。

则可立即得定理，当 $n \geq 1$，则有

$$\sum_{d|n} \mu(d) = [\frac{1}{n}] \tag{1}$$

Möbius 变换：

$$\text{eg1. } n = \sum_{d|n} \varphi(d) = \sum_{d|n} \varphi(\frac{n}{d}) \tag{2}$$

$$\text{eg2. } \varphi(n) = \sum_{d|n} \mu(d)\frac{n}{d} = \sum_{d|n} \mu(\frac{n}{d})d \tag{3}$$

定义：若数论函数 $f(n)$ 和 $g(n)$ 适合

$$f(n)=\sum_{d|n}g(d)=\sum_{d|n}g(\frac{n}{d}) \tag{4}$$

则称 $f(n)$ 为 $g(n)$ 的 Möbius 变换，而 $f(n)$ 为 $g(n)$ 的 Möbius 逆变换。

定理：若任意两个数论函数 $f(n)$ 和 $g(n)$ 满足等式

$$f(n)=\sum_{d|n}g(d) \tag{5}$$

则有

$$g(n)=\sum_{d|n}\mu(d)f(\frac{n}{d}) \tag{6}$$

反之也成立。

4.5.1 递推法

Möbius 函数有一很好的性质：$\sum_{d|n}\mu(d)=[n=1]$，因而可以递推求 Möbius 函数。

复杂度	$O(N\log N)$	
全局	mu[]	Möbius 函数的值

```
const int MaxN = 1000005;
int64 mu[MaxN];

void getMu()
{
    for (int i = 1; i < MaxN; i++)
    {
        int target = (i == 1)? 1: 0;
        int delta = target - mu[i];
        mu[i] = delta;
        for (int j = i + i; j < MaxN; j += i)
            mu[j] += delta;
    }
}
```

4.5.2 线性筛

类似素数的线性筛法，可以将计算 Möbius 函数的筛法优化至线性时间复杂度。即在筛素数的同时，用递推式的结论，将其计算。

复杂度	$O(N)$	
全局	isPrime[]	是否为素数
	prime[]	素数表，从下标 0 开始
	tot	素数个数
	mu[]	Möbius 函数的值

```
const int MaxN = 1000005;
bool isPrime[MaxN];
int tot, prime[MaxN], mu[MaxN];

void getMu_Prime()
{
    fill(isPrime, isPrime + MaxN, true);
    isPrime[0] = isPrime[1] = 0;
```

```
    mu[1] = 1; tot = 0;
    for (int i = 2; i < MaxN; i++)
    {
        if (isPrime[i]) { prime[tot++]; phi[i] = -1; }
        for (int j = 0; j < tot; j++)
        {
            if (i * prime[j] >= MaxN) break;
            isPrime[i * prime[j]] = false;
            if (i % prime[j] != 0)
                mu[i * prime[j]] = -mu[i];
            else { mu[i * prime[j]] = 0; break; }
        }
    }
}
```

4.5.3 例子

这里给出几个 Möbius 反演的例子，用于展示它的优越性。

例子 1	求区间的两个数互素的数目	calc(int a, int b)
例子 2	求区间的三个数互素的数目	calc(int a, int b, int c)
例子 3	求区间的两个数 $gcd = d$ 的数目	find(int a, int b)
例子 4	求区间的三个数 $gcd = d$ 的数目	find(int a, int b, int c)
例子 5	求在三个方向分别有 a, b, c 个整点的长方体从一个顶点能够看到的整点数	looking(int a, int b, int c)

```
int summ[MaxN];
void getPrefixSum()
{
    summ[0] = 0;
    for (int i = 1; i < MaxN; i++)
        summ[i] = summ[i - 1] + mu[i];
}

// calculate the pairs of (i, j) when gcd(i, j) = 1
// which (i, j) is in grid [1..a][1..b]
int64 calc(int a, int b)
{
    int n = min(a, b), d1, d2, n1, n2, nn;
    int64 ans = 0;
    for (int i = 1; i <= n; i = nn + 1)
    {
        d1 = a / i;  d2 = b / i;
        n1 = a / d1; n2 = b / d2;
        nn = min(n1, n2);
        ans += (int64) d1 * d2 * (summ[nn] - summ[i - 1]);
    }
    return ans;
}

// calculate the pairs of (i, j, k) when gcd(i, j, k) = 1
// which (i, j, k) is in cube [1..a][1..b][1..c]
int64 calc(int a, int b, int c)
{
    int n = min(min(a, b), c);
    int d1, d2, d3, n1, n2, n3, nn;
    int64 ans = 0;
    for (int i = 1; i <= n; i = nn + 1)
    {
        d1 = a /  i; d2 = b /  i; d3 = c /  i;
        n1 = a / d1; n2 = b / d2; n3 = c / d3;
        nn = min(min(n1, n2), n3);
        ans += (int64) d1 * d2 * d3 * (summ[nn] - summ[i - 1]);
    }
    return ans;
```

```
}

// calculate the pairs of (i, j) when gcd(i, j) = d
// which (i, j) is in grid [1..a][1..b]
int64 find(int a, int b, int d)
{
    if (a == 0 || b == 0 || d == 0) return 0;
    else return calc(a / d, b / d);
}

// calculate the pairs of (i, j, k) when gcd(i, j, k) = d
// which (i, j, k) is in cube [1..a][1..b][1..c]
int64 find(int a, int b, int c, int d)
{
    if (a == 0 || b == 0 || c == 0 || d == 0) return 0;
    else return calc(a / d, b / d, c / d);
}

// calculate the points of a cuboid which has a, b, c points
// in x, y, z directions, when looking from point(0, 0, 0)
int64 looking(int a, int b, int c)
{
    a--; b--; c--;
    return 3 + calc(a, b, c) + calc(a, b) + calc(a, c) + calc(b, c);
}
```

5 数值计算

5.1 浮点数二分计算

二分法只可以求解单增或单减的函数零点。

复杂度	$O(\log K)$	
输入	l	左界
	r	右界
	ans	返回零点

```
const double eps = 1e-15;
inline bool test(double x)
{
    // true : l = mid
    // false: r = mid
    /**Specific Calculation**/
}
double Bsearch(double l, double r)
{
    while (r - l > eps)
    {
        double mid = (r + l) / 2;
        if (test(mid)) l = mid;
        else r = mid;
    }
    return tmp;
}
```

5.2 浮点数三分计算

三分法只可以求单峰或单谷的函数极值点。

5.2.1 等分法

三等分法选取参照点。

复杂度	$O(\log K)$	
输入	l	左界
	r	右界
	ans	返回极值点

```
// <!-- encoding UTF-8 --!>
/*******************************************************************************
*                      ----Stay Hungry Stay Foolish----                        *
*    @author    :   Shen                                                      *
*    @name      :   ZOJ 3203                                                  *
*******************************************************************************/
#include <bits/stdc++.h>
using namespace std;
typedef long long int64;
template<class T>inline bool updateMin(T& a, T b){ return a > b ? a = b, 1: 0; }
template<class T>inline bool updateMax(T& a, T b){ return a < b ? a = b, 1: 0; }
inline int    nextInt() { int x; scanf("%d", &x); return x; }
inline int64  nextI64() { int64  d; cin >> d; return d; }
inline char   nextChr() { scanf(" "); return getchar(); }
inline string nextStr() { string s; cin >> s; return s; }
inline double nextDbf() { double x; scanf("%lf", &x); return x; }
inline int64  nextlld() { int64 d; scanf("%lld", &d); return d; }
inline int64  next64d() { int64 d; scanf("%I64d",&d); return d; }

```

```
const double eps = 1e-9;
double D, H, h;

inline double calcf(double x)
{
    return D - x + H - (H - h) * D / x;
}

inline bool test(double xl, double xr)
{
    // true : l = mid
    // false: r = midmid
    /**Specific Calculation**/
    return calcf(xl) < calcf(xr);
}

double Tsearch_e(double l, double r)
{
    ///@return the x, not the f(x)
    double midl = 0, midr = 0;
    while (r - l > eps)
    {
        midl = (2 * l + r) / 3;
        midr = (2 * r + l) / 3;
        if (test(midl, midr)) l = midl;
        else r = midr;
    }
    return midl;
}

void solve()
{
    H = nextDbf(); h = nextDbf(); D = nextDbf();
    double x = Tsearch_e((H - h) * D / H, D);
    printf("%.3lf\n", calcf(x));
}

int main()
{
    int t = nextInt(); while (t--) solve();
    return 0;
}
```

5.2.2 midmid 法

midmid 法选取参照点，即同时做两个操作，$mid := (r+l)/2$ 和 $mid := (r+mid)/2$。来选取参照点。

复杂度	$O(\log K)$	
输入	l	左界
	r	右界
	ans	返回极值点

```
// <!-- encoding UTF-8 --!>
/*************************************************************************
*                      ----Stay Hungry Stay Foolish----                  *
*    @author    :   Shen                                                 *
*    @name      :   ZOJ 3203                                             *
*************************************************************************/
#include <bits/stdc++.h>
using namespace std;
typedef long long int64;
template<class T>inline bool updateMin(T& a, T b){ return a > b ? a = b, 1: 0; }
template<class T>inline bool updateMax(T& a, T b){ return a < b ? a = b, 1: 0; }
inline int    nextInt() { int x; scanf("%d", &x); return x; }
inline int64  nextI64() { int64  d; cin >> d; return d; }
```

```
inline char   nextChr() { scanf(" "); return getchar(); }
inline string nextStr() { string s; cin >> s; return s; }
inline double nextDbf() { double x; scanf("%lf", &x); return x; }
inline int64  nextlld() { int64 d; scanf("%lld", &d); return d; }
inline int64  next64d() { int64 d; scanf("%I64d",&d); return d; }

const double eps = 1e-9;
double D, H, h;

inline double calcf(double x)
{
    return D - x + H - (H - h) * D / x;
}

inline bool test(double xl, double xr)
{
    // true : l = mid
    // false: r = midmid
    /**Specific Calculation**/
    return calcf(xl) < calcf(xr);
}

double Tsearch(double l, double r)
{
    ///@return the x, not the f(x)
    double mid = 0, midmid = 0;
    while (r - l > eps)
    {
        mid = (r + l) / 2;
        midmid = (mid + r) / 2;
        if (test(mid, midmid)) l = mid;
        else r = midmid;
    }
    return mid;
}

void solve()
{
    H = nextDbf(); h = nextDbf(); D = nextDbf();
    double x = Tsearch((H - h) * D / H, D);
    printf("%.3lf\n", calcf(x));
}

int main()
{
    int t = nextInt(); while (t--) solve();
    return 0;
}
```

5.2.3 优选法

优选法选取参照点，即同时选取两个黄金分割点来作为参照点。这样能够减少一次运算，同时迭代次数更稳定。

复杂度	$O(\log K)$	
输入	l	左界
	r	右界
	ans	返回极值点

```
// <!-- encoding UTF-8 --!>
/*************************************************************************
*                      ----Stay Hungry Stay Foolish----                  *
*    @author    :   Shen                                                 *
*    @name      :   ZOJ 3203                                             *
*************************************************************************/
```

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long int64;
template<class T>inline bool updateMin(T& a, T b){ return a > b ? a = b, 1: 0; }
template<class T>inline bool updateMax(T& a, T b){ return a < b ? a = b, 1: 0; }
inline int    nextInt() { int x; scanf("%d", &x); return x; }
inline int64  nextI64() { int64  d; cin >> d; return d; }
inline char   nextChr() { scanf(" "); return getchar(); }
inline string nextStr() { string s; cin >> s; return s; }
inline double nextDbf() { double x; scanf("%lf", &x); return x; }
inline int64  nextlld() { int64 d; scanf("%lld", &d); return d; }
inline int64  next64d() { int64 d; scanf("%I64d",&d); return d; }

const double eps = 1e-9;
const double cef = (sqrt(5.0) - 1.0) * 0.5;
double D, H, h;

inline double calcf(double x)
{
    return D - x + H - (H - h) * D / x;
}

inline bool test(double xl, double xr)
{
    // true : l = mid
    // false: r = midmid
    /**Specific Calculation**/
    return calcf(xl) < calcf(xr);
}

double Tsearch_s(double l, double r)
{
    ///@return the x, not the f(x)
    double midl = r - (r - l) * cef;
    double midr = l + (r - l) * cef;
    while (r - l > eps)
    {
        if (test(midl, midr))
        {
            l = midl; midl = midr;
            midr = l + (r - l) * cef;
        }
        else
        {
            r = midr; midr = midl;
            midl = r - (r - l) * cef;
        }
    }
    return midr;
}

void solve()
{
    H = nextDbf(); h = nextDbf(); D = nextDbf();
    double x = Tsearch_s((H - h) * D / H, D);
    printf("%.3lf\n", calcf(x));
}

int main()
{
    int t = nextInt(); while (t--) solve();
    return 0;
}
```

5.3 数值积分

5.3.1 Simpson 方法

利用二次曲线逼近法来计算函数积分。

复杂度	$O(N)$	
输入	f	函数 f
	a	积分下限
	b	积分上限
	n	均分份数
输出	double	$\int_a^b f(x)dx$

```
template<class T>
double simpson(const T& f, double a, double b, int n)
{
    const double h = (b - a) / n;
    double ans = f(a) + f(b);
    for (int i = 1; i < n; i += 2) ans += 4 * f(a + i * h);
    for (int i = 2; i < n; i += 2) ans += 2 * f(a + i * h);
    return ans * h / 3;
}
```

5.3.2 Romberg 方法

利用 Romberg 方法来计算函数积分，其误差阶是 $O(h^8)$。

复杂度	$O(\log^2 K)$	
输入	f	函数 f
	a	积分下限
	b	积分上限
	eps	允许误差
输出	double	$\int_a^b f(x)dx$

```
template<class T>
double romberg(const T& f, double a, double b, double eps = 1e-8)
{
    vector<double> t; double h = b - a, last, curr;
    int k = 1, i = 1;
    t.push_back(h * (f(a) + f(b)) / 2);
    while (true)
    {
        last = t.back(); curr = 0;
        double x = a + h / 2;
        for (int j = 0; j < k; j++, x += h) curr += f(x);
        curr = (t[0] + h * curr) / 2;
        double k1 = 4.0 / 3.0, k2 = 1.0 / 3.0;
        for (int j = 0; j < i ;j++)
        {
            double temp = k1 * curr - k2 * t[j];
            t[j] = curr; curr = temp;
            k2 /= 4 * k1 - k2; k1 = k2 + 1;
        }
        t.push_back(curr);
        k *= 2; h /= 2; i++;
        if (fabs(last - curr) < eps) break;
    }
    return t.back();
}
```

5.4 高阶方程求根

对一给定方程 $a_n x^n + a_{n-1} x^{n-1} + \cdot + a_1 x + a_0 = 0$ ，求出该方程的所有实数解。

对于五次方程以下的代数方程的公式解不在讨论范围。五次与五次以上的代数方程没有代数根（方程根式有解定理）。

对于一般的 n 次方程，首先对其求导，然后求出所有导函数的所有零点。那么在导函数的两个零点之间，该 n 次方程必然是单调的，并且最多只有一个零点。利用此性质，可以用二分法求出这个零点。

对于求解导函数的零点问题可以递归求解，直到为一元一次方程为止。

需要注意的是，该算法由于无法使用数组（递归性），所以 vector 无疑增加了时间开销，同时，递归调用也使得此算法整体的时间效率不佳，故需酌情使用。

复杂度	$O(N^2 \log K)$	
输入	coef	方程系数，$coef[i] = a_i$
	n	方程的次数
输出	vector<double>	所有实数解

```
const double eps = 1e-12;
const double inf = 1e+12;

inline int sign(double x) { return (x < -eps)? -1 : x > eps; }

inline double get(const vector<double>& coef, double x)
{
    double e = 1, s = 0; int sz = coef.size();
    for (int i = 0; i < sz; i++, e *= x) s += coef[i] * e;
    return s;
}

double find(const vector<double>& coef, int n, double lo, double hi)
{
    double sign_lo, sign_hi;
    if ((sign_lo = sign(get(coef, lo))) == 0) return lo;
    if ((sign_hi = sign(get(coef, hi))) == 0) return hi;
    if (sign_lo * sign_hi > 0) return inf;
    for (int step = 0; step < 100 && hi - lo > eps; step++)
    {
        double m = (lo + hi) * 0.5;
        int sign_mid = sign(get(coef, m));
        if (sign_mid == 0) return m;
        if (sign_lo * sign_mid < 0) hi = m;
        else lo = m;
    }
    return (lo + hi) * 0.5;
}

vector<double> solve(vector<double> coef, int n)
{
    vector<double> ret;
    if (n == 1)
    {
        if (sign(coef[1])) ret.push_back(-coef[0] / coef[1]);
        return ret;
    }
    vector<double> dcoef(n);
    for (int i = 0; i < n; i++) dcoef[i] = coef[i + 1] * (i + 1);
    vector<double> droot = solve(dcoef, n - 1);
    droot.insert(droot.begin(), -inf);
    droot.push_back(inf);
    for (int i = 0; i + 1 < droot.size(); i++)
    {
        double tmp = find(coef, n, droot[i], droot[i + 1]);
        if (tmp < inf) ret.push_back(tmp);
    }
    return ret;
}
```

5.5 快速傅里叶变换

普通的多项式乘法的时间复杂度基本为 $O(N^2)$，不过却有一个更快的方法，那就是利用快速傅里叶变换，使得时间复杂度优化至 $(N\log N)$。

如果我们把多项式看做一个向量形式，即只考虑多项式的系数，那么，多项式的乘法即相当于求解响亮的卷积。一个更好的思路是，将两个多项式转化为点值表达。

通俗的讲。对于多项式

$$A(x)=a_0+a_1x+\cdot+a_{n-1}x^{n-1}+a_nx^n \tag{7}$$

它的向量表达是 $(a_0,a_1,\cdot,a_{n-1},a_n)$，如果这个多项式经过 n 个不同的点，即 $y_k=A(x_k),x_i\neq x_j$，则这些点所能构成的一个集合

$$\{(x_0,y_0),(x_1,y_1),\cdot,(x_{n-1},y_{n-1})\} \tag{8}$$

该集合也被称为点值表达。

对于一个多项式，可以通过计算出点值表达，也可以通过进行多项式插值，得到原多项式。

下面说明如何利用傅里叶变换求多项式乘法。

众所周知，若多项式 $A(x)$ 和 $B(x)$ 的点值表达分别为

$$A(x):=\{(x_0,y_0),(x_1,y_1),\cdot,(x_{n-1},y_{n-1})\} \tag{9}$$

与

$$B(x):=\{(x_0,y_0{}'),(x_1,y_1{}'),\cdot,(x_{n-1},y_{n-1}{}')\} \tag{10}$$

那么记多项式 $C(x)=A(x)B(x)$，则显然，$\partial(C(x))=\partial(A(x))+\partial(B(x))$，

所以，如果多项式 $A(x)$ 和 $B(x)$ 的点值表达的点值数目达到 $\partial(A(x))+\partial(B(x))$ 就可以得到

$$A(x):=\{(x_0,y_0),(x_1,y_1),\cdot,(x_{2n-1},y_{2n-1})\} \tag{11}$$

与

$$B(x):=\{(x_0,y_0{}'),(x_1,y_1{}'),\cdot,(x_{2n-1},y_{2n-1}{}')\} \tag{12}$$

故有：

$$C(x):=\{(x_0,y_0y_0{}'),(x_1,y_1y_1{}'),\cdot,(x_{2n-1},y_{2n-1}y_{2n-1}{}')\} \tag{13}$$

此时可以用多项式插值求出其向量表达。

总结如下：

(1) 向量形式求值，得到长度为 2n 的点值列，时间复杂度 $O(N\log N)$；

(2) 点值乘法，得到 $C(x)$ 的点值表达，时间复杂度 $O(N)$；

(3) 多项式插值，得到长度为 2n 的向量形式，时间复杂度 $O(N\log N)$。

另外选择单位复数根作为求值点，并且通过分治加速傅里叶变换，就可以得到时间复杂度为 $(N\log N)$ 的计算多项式乘法的高效算法了。

具体的实现利用到了一种“蝴蝶操作”，其简要原理如下：

$$\begin{aligned} y_k^{[0]} &\to y_k^{[0]}+\omega_n^k y_k^{[1]} \\ y_k^{[1]} &\to y_k^{[0]}-\omega_n^k y_k^{[1]} \end{aligned} \tag{14}$$

至此，该算法的时间复杂度已经被优化至近似为 $O(N\log N)$。

另外有几处使用说明：

(1) 保证高位有足够的 0；

(2) FFT 要求 len 必须为二的幂次，所以必须补齐 0；

(3) DFT 要求是定义在复数上的，所以有与证书的变换要求；

(4) 高精度乘法需要在多项式乘法的基础上实现进位。

复杂度	$O(N\log N)$		

输入	Complex y[]	需要变换的数据
	len	数据大小
	on	开关，1 表示 DFT，-1 表示 iDFT

下面给出两个例子，hdu 1402 是利用 FFT 快速的求大数乘法，hdu 4609 是利用 FFT 快速的求选择方案的总数。

5.5.1 hdu 1402

将数字看做以十为基的多项式，然后类似多项式乘法的操作处理即可。最后处理进位。

```
// <!--encoding UTF-8 UTF编码-8--!>
/*************************************************************************
*                     ----Stay Hungry Stay Foolish----                   *
*    @author    :   Shen                                                 *
*    @name      :   HDU 1402                                             *
*************************************************************************/

#include <map>
#include <list>
#include <queue>
#include <stack>
#include <cmath>
#include <vector>
#include <string>
#include <cstdio>
#include <cstring>
#include <cstdlib>
#include <iostream>
#include <algorithm>
using namespace std;
typedef long long int64;
template<class T>inline bool updateMin(T& a, T b){ return a > b ? a = b, 1: 0; }
template<class T>inline bool updateMax(T& a, T b){ return a < b ? a = b, 1: 0; }
inline int    nextInt() { int x; scanf("%d", &x); return x; }
inline int64  nextI64() { int64  d; cin >> d; return d; }
inline char   nextChr() { scanf(" "); return getchar(); }
inline string nextStr() { string s; cin >> s; return s; }
inline double nextDbf() { double x; scanf("%lf", &x); return x; }
inline int64  nextlld() { int64 d; scanf("%lld", &d); return d; }
inline int64  next64d() { int64 d; scanf("%I64d",&d); return d; }

const double PI = acos(-1.0);
struct Complex
{
    double x, y; // z = x + iy
    Complex(double _x = 0.0, double _y = 0.0) { x = _x; y = _y; }
    Complex operator+(const Complex& b) const
    {
        return Complex(x + b.x, y + b.y);
    }
    Complex operator-(const Complex& b) const
    {
        return Complex(x - b.x, y - b.y);
    }
    Complex operator*(const Complex& b) const
    {
        return Complex(x * b.x - y * b.y, x * b.y + y * b.x);
    }
};

// the flip operation before FFT & iFFT
void change(Complex y[], int len)
{
    for (int i = 1, j = len / 2; i < len - 1; i++)
    {
        if (i < j) swap(y[i], y[j]);
```

```
        int k = len / 2;
        while (j >= k) { j -= k; k /= 2; }
        if (j < k) j += k;
    }
}

// fft operation, len format like 2^k
// do DFT when on = 1, iDFT when on = -1
void fft(Complex y[], int len, int on)
{
    change(y, len);
    for (int h = 2; h <= len; h <<= 1)
    {
        Complex wn(cos(-on * 2 * PI / h), sin(-on * 2 * PI / h));
        for (int j = 0; j < len; j += h)
        {
            Complex w(1, 0);
            for (int k = j; k < j + h / 2; k++)
            {
                Complex u = y[k], t = w * y[k + h / 2];
                y[k] = u + t; y[k + h / 2] = u - t;
                w = w * wn;
            }
        }
    }
    if (on == -1) for (int i = 0; i < len; i++)
        y[i].x /= len;
}

const int MaxN = 200010;
Complex x1[MaxN], x2[MaxN];
char str1[MaxN / 2], str2[MaxN / 2];
int sum[MaxN];

void solve()
{
    int len1 = strlen(str1), len2 = strlen(str2);
    int len = 1;
    while(len < len1*2 || len < len2*2)len<<=1;
    for(int i = 0; i < len1; i++)
        x1[i] = Complex(str1[len1 - 1 - i] - '0',0);
    for(int i = len1; i < len; i++)
        x1[i] = Complex(0, 0);
    for(int i = 0; i < len2; i++)
        x2[i] = Complex(str2[len2 - 1 - i] - '0',0);
    for(int i = len2; i < len; i++)
        x2[i] = Complex(0, 0);
    // DFT
    fft(x1, len, 1); fft(x2, len, 1);
    for (int i = 0; i < len; i++) x1[i] = x1[i] * x2[i];
    fft(x1, len, -1);
    for (int i = 0; i < len; i++)
        sum[i] = (int)(x1[i].x + 0.5); // round
    for (int i = 0; i < len; i++)
    {
        sum[i + 1] += sum[i] / 10;
        sum[i] %= 10;
    }
    len = len1 + len2 - 1;
    while (sum[len] <= 0 && len > 0) len--;
    for (int i = len; i >= 0; i--) printf("%c", sum[i] + '0');
    puts("");
}

int main()
{
    while (~scanf("%s%s", str1, str2)) solve();
    return 0;
```

```
125 }
```

5.5.2 hdu 4609

将长度不同木条数据组看做相同元的不同次数的表示形式，比如样例的 $\{1,3,3,4\}$ 可以看做向量 $(0,1,0,2,1)$，表示长度为 0 的有 0 根，长度为 1 的有 1 根，长度为 2 的有 0 根，长度为 3 的有 2 根，长度为 4 的有 1 根。

那么选取的结果就相当于向量自身的卷积，即 $(0,1,0,2,1)\bigotimes(0,1,0,2,1) = (0,0,1,0,4,2,4,4,1)$ 表示和为 2 的取法为 1 种，和为 4 的取法为 4 种，和为 5 的取法为 2 种，和为 6 的取法为 4 种，和为 7 的取法为 4 种，和为的 8 取法为 1 种。

随后便是删除修饰工作：

(1) 自身组合不行，所以删除自身的组合；

(2) 选取没有先后顺序，所以除以 2；

针对每一种可能：

(3) 减去一个取大的，一个取小的

(4) 减去取本身的

(5) 减去大于它的取的组合

```
1  // <!--encoding UTF-8 UTF编码-8--!>
2  /*******************************************************************************
3  *                        ----Stay Hungry Stay Foolish----                      *
4  *    @author    :   Shen                                                       *
5  *    @name      :   HDU 4609                                                   *
6  *******************************************************************************/
7
8  #include <map>
9  #include <list>
10 #include <queue>
11 #include <stack>
12 #include <cmath>
13 #include <vector>
14 #include <string>
15 #include <cstdio>
16 #include <cstring>
17 #include <cstdlib>
18 #include <iostream>
19 #include <algorithm>
20 using namespace std;
21 typedef long long int64;
22 template<class T>inline bool updateMin(T& a, T b){ return a > b ? a = b, 1: 0; }
23 template<class T>inline bool updateMax(T& a, T b){ return a < b ? a = b, 1: 0; }
24 inline int    nextInt() { int x; scanf("%d", &x); return x; }
25 inline int64  nextI64() { int64  d; cin >> d; return d; }
26 inline char   nextChr() { scanf(" "); return getchar(); }
27 inline string nextStr() { string s; cin >> s; return s; }
28 inline double nextDbf() { double x; scanf("%lf", &x); return x; }
29 inline int64  nextlld() { int64 d; scanf("%lld", &d); return d; }
30 inline int64  next64d() { int64 d; scanf("%I64d",&d); return d; }
31
32 const double PI = acos(-1.0);
33 struct Complex
34 {
35     double x, y; // z = x + iy
36     Complex(double _x = 0.0, double _y = 0.0) { x = _x; y = _y; }
37     Complex operator+(const Complex& b) const
38     {
39         return Complex(x + b.x, y + b.y);
40     }
41     Complex operator-(const Complex& b) const
42     {
43         return Complex(x - b.x, y - b.y);
44     }
45     Complex operator*(const Complex& b) const
46     {
```

```
47          return Complex(x * b.x - y * b.y, x * b.y + y * b.x);
48      }
49  };
50
51  // the flip operation before FFT & iFFT
52  void change(Complex y[], int len)
53  {
54      for (int i = 1, j = len / 2; i < len - 1; i++)
55      {
56          if (i < j) swap(y[i], y[j]);
57          int k = len / 2;
58          while (j >= k) { j -= k; k /= 2; }
59          if (j < k) j += k;
60      }
61  }
62
63  // fft operation, len format like 2^k
64  // do DFT when on = 1, iDFT when on = -1
65  void fft(Complex y[], int len, int on)
66  {
67      change(y, len);
68      for (int h = 2; h <= len; h <<= 1)
69      {
70          Complex wn(cos(-on * 2 * PI / h), sin(-on * 2 * PI / h));
71          for (int j = 0; j < len; j += h)
72          {
73              Complex w(1, 0);
74              for (int k = j; k < j + h / 2; k++)
75              {
76                  Complex u = y[k], t = w * y[k + h / 2];
77                  y[k] = u + t; y[k + h / 2] = u - t;
78                  w = w * wn;
79              }
80          }
81      }
82      if (on == -1) for (int i = 0; i < len; i++)
83          y[i].x /= len;
84  }
85
86  const int MaxN = 400040;
87  Complex x1[MaxN];
88  int a[MaxN / 4];
89  int64 num[MaxN], sum[MaxN];
90
91  void solve()
92  {
93      int n = nextInt(); memset(num, 0, sizeof(num));
94      for (int i = 0; i < n; i++)
95      {
96          a[i] = nextInt();
97          num[a[i]]++;
98      }
99      sort(a, a + n);
100     int len1 = a[n - 1] + 1, len = 1;
101     while (len < 2 * len1) len <<= 1;
102     for (int i = 0; i < len1; i++)
103         x1[i] = Complex(num[i], 0);
104     for (int i = len1; i < len; i++)
105         x1[i] = Complex(0, 0);
106     // DFT
107     fft(x1, len, 1);
108     for (int i = 0; i < len; i++) x1[i] = x1[i] * x1[i];
109     fft(x1, len, -1);
110     for (int i = 0; i < len; i++)
111         num[i] = (int64)(x1[i].x + 0.5); // round
112     len = 2 * a[n - 1];
113   // delete the same choice
114     for (int i = 0; i < n; i++) num[a[i] + a[i]]--;
```

```
115   // orderless choosing, divides by 2
116     for (int i = 1; i <= len; i++) num[i] /= 2;
117     int64 cnt = 0; sum[0] = 0;
118   // calc prefix sum
119     for (int i = 1; i <= len; i++) sum[i] = sum[i - 1] + num[i];
120     for (int i = 0; i < n; i++)
121     {
122     cnt += sum[len] - sum[a[i]];
123     // choosed one too big and one too small
124         cnt -= (int64)(n - 1 - i) * i;
125     // includes i-self
126         cnt -= (n - 1);
127     // choosed both too big
128         cnt -= (int64)(n - 1 - i) * (n - i - 2) / 2;
129     }
130     int64 tot = (int64)n * (n - 1) * (n - 2) / 6;
131     printf("%.7lf\n", (double)cnt / tot);
132 }
133
134 int main()
135 {
136     int t = nextInt(); while (t--) solve();
137     return 0;
138 }
```

6 其他

6.1 进制转换

把一个 x 进制的数转换成 y 进制。
具体做法是先把 x 进制转换为 10 进制，然后在不断地取模倒序转换成 y 进制。
若高进制的字母表示有区别，请注意同时修改两处字母。

复杂度	$O(L)$	
输入	x	原数据进制数，$2 \leq x \leq 62$
	y	新数据进制数，$2 \leq x \leq 62$
	s	原数据的字符串形式
输出	string	新数据的字符串形式

```
string transform(int x, int y, string s)
{
    int sz = s.size(), sum = 0; string res = "";
    for (int i = 0; i < sz; i++)
    {
        if (s[i] == '-') continue;
        if (s[i] >= '0' && s[i] <= '9')
            sum = sum * x + s[i] - '0';
        else if (s[i] >= 'A' && s[i] <= 'Z')
            sum = sum * x + s[i] - 'A' + 10;
        else sum = sum * x + s[i] - 'a' + 10 + 26;
    }
    while (sum)
    {
        char tmp = sum % y; sum /= y;
        if (tmp <= 9) tmp += '0';
        else if (tmp <= 36) tmp += 'A' - 10;
        else tmp += 'A' - 10 - 26;
        res = tmp + res;
    }
    if (res.size() == 0) res = "0";
    if (s[0] == '-') res = '-' + res;
    return res;
}
```

6.2 格雷码

给一个二进制的位数 n，求出一个 0 到 $2^n - 1$ 的排列，使得相邻两项（包括首尾相邻）的二进制表达中只有恰好以为不同。

由数学知识可知，一种简单的格雷码编码方式有规律如下：

$g[i] = ixor(i >> 1)$

复杂度	$O(2^n)$	
输入	n	二进制的位数
输出	vector<int>	n 位的格雷码序列

```
vector<int> initGray(int n)
{
    vector<int> res; res.clear();
    for (int i = 0; i < (1 << n); i++)
        res.push_back(i ^ (i >> 1));
    return res;
}
```