24th BULGARIAN NATIONAL OLYMPIAD IN INFORMATICS

Solutions of selected problems

telerik
deliver more than expected

2008

Round 1

Round 1 / Task A1. FRACTIONS

The fractions that we are going to use can be considered as elements of a structure with two members, *p* and *q*, denoting the fraction numerator and denominator, respectively. To solve the task, we generate a fraction $\frac{a}{b}$ for which $k \le a < b \le n$, check if it is non-reducible and if it is, we add it to the current values of variable *s* which we use to calculate to final sum. As initial values of *s.p* and *s.q* we assign 0 and 1, respectively. After each addition of two fractions we make possible reductions.

```
#include<iostream>
using namespace std;

struct Fract
{ int p,q; };

int nod(int a, int b)
{ int r = a%b;
  while(r > 0)
  { a = b; b = r; r = a%b; }
  return b;
}

int main()
{   int k, n;
    cin >> k >> n;
    Fract s, x;
    s.p = 0;
    s.q = 1;
    for(int a=k; a<n; a++)
      for(int b=a+1; b<=n; b++)
        if (nod(a,b) == 1)
          {x.p = a;
           x.q = b;
           int p = s.p*x.q + s.q*x.p;
           int q = s.q*x.q;
           int d = nod(p,q);
           s.p = p/d;
           s.q = q/d;
          }
    cout << s.p << "/" << s.q << endl;
    return 0;
}
```

Round 1 / Task A3. BELONGING

The limits allow direct verification for each of the input values, using the rules. 18 decimal digits fit in an 8-byte integer type. Such approach risks being a little slow for some big integers (**`inM1`** in the realization).

Here are some properties of the binary representation for each member of ***M*** that follow from the rules:

- The total count of the binary digits is odd. Sure enough: 1 is with one digit; rule 2 adds two more binary digits, so it doesn't change the parity, and rule 3 combines two previous lengths, adding one more digit. As we start at one digit (odd) there is no rule to change the parity of the total binary digits count – it remains odd.
- Rules don't arouse numbers with adjacent zeroes. Moreover, zeroes can only occupy an even place, according to rule 2.
- All numbers with odd total length of the binary representation and zeroes (if any) at even places are members of ***M***. For length of 1 this is trivial (rule 1). Let's assume it true for all odd lengths less than or equal to the odd integer ***n***. Consider now the numbers with binary representation size of ***n***+2. If there is no zero in it, it is given rise by the number 1 (rule 1), applying (***n***+1)/2 times rule 2. If there is (at least) one zero (at an even place, of course), on its left and on its right there are representations with odd lengths, less or equal to ***n***, and zeroes (if any) at even places. Using the inductive assumption there can be all representations with corresponding lengths there. Rule 3 states that such combination belongs to ***M***.

The use of these facts leads to a linear with regard to the binary digits count (i.e., logarithmic in regard to the candidate) algorithm for determining the membership to ***M***. (function **`inM2`**).

Realization

```
#include <iostream>
#include <stdlib.h>
#include <string.h>
using namespace std;
long long N=5, P, Q;
int rec(char *s,int l)
{if (l<=0) return 0;
 if (l==1) return *s=='1';
 if (s[0]=='1' && s[1]=='1' && rec(&s[2],l-2)) return 1;
 for (int i=1;i<l-1;i++)
  if (s[i]=='0' && rec(s,i) && rec(&s[i+1],l-i-1)) return 1;
 return 0;
}
char *strrev(char *s)
{char c;
 for (int i=0,j=strlen(s)-1;i<j;i++,j--)
 {c=s[i];s[i]=s[j];s[j]=c;}
 return s;
}
```

```
int inM1(long long a)
{char b[64],i;
 for (i=0;a;i++)
  {b[i]='0'+ (a&1); a>>=1;}
 b[i]=0;
 strrev(b);
 return rec(b,strlen(b));
}
int inM2(long long a)
{char d[64],c=0;
 do
 {d[c++]=a&1;
  a>>=1;
 }while (a);
 if (!(c&1)) return 0;
 for (c--;c>=0;c-=2) if (!d[c]) return 0;
 return 1;
}
int main()
{cin>>N>>P>>Q;
 //cout<<inM1(N)<<inM1(P)<<inM1(Q)<<endl;
 cout<<inM2(N)<<inM2(P)<<inM2(Q)<<endl;
 return 0;
}
```

Round 1 / Task B1. MATH

The *n* numbers on the board can be considerd as elements of one-dimensional array *a*. The maximum sum can be obtained if the subtraction signs are placed before the smallest elements of the array. As we cannot put a sign before the first number we have to find the smallest integers, *k* in number, among all numbers on the board except the first one. To solve the problem, we can do the following:

1. Sort in a descending order the elements of the array, without $a[0]$.
2. Find the sum of the elements of the array from $a[0]$ to $a[n-k-1]$.
3. Subtract from this sum the remaining *k* elements of the array.

```
#include<iostream>
#include<algorithm>

using namespace std;

bool comp(int x, int y)
{ return x > y; }

int main()

{ int n, k;
  cin >> n >> k;
  int a[32];
  for(int i=0; i<n; i++)
```

```
    cin >> a[i];
  sort(a+1, a+n, comp);
  int s = a[0];
  for(int i=1; i<=n-k-1; i++)
    s = s + a[i];
  for(int i=n-k; i<n; i++)
    s = s - a[i];
  cout << s << endl;
  return 0;
}
```

Round 1 / Task B2. SUM OF DIGITS

Linear scanning (function **`sum`**) works fast enough for these limitations, if well designed. The sum of the digits in the binary representation is actually the count of the ones in it. We can use the well-known algorithm which implements a bitwise AND to find this count (function **`count1`** in the realization).

For bigger numbers the following improvement can be used.

It is easy to realize that if $\boldsymbol{n}$ is with one less than a power of two, i.e. $\boldsymbol{n}=2^{\boldsymbol{k}}-1$, its binary representation consists of ones only (***k*** times), and together with the previous numbers they represent, we may say, all the subsets of a ***k***-element set. Thus ones will be met $2^{\boldsymbol{k}-1}$ times in each of ***k*** places, which means that the desired sum up to any such $\boldsymbol{n}$ will be $\boldsymbol{k}.2^{\boldsymbol{k}-1}$. This idea can be developed further but in the realization it is used only until here: we find the biggest integer of the form $2^{\boldsymbol{k}}-1$ (i.e., containing only ones in its binary representation) not exceeding ***n***, we calculate the sum up to there using the formula above, and scan the rest numbers up to ***n*** (function **`sum1`**).

Realization

```
#include <iostream>
using namespace std;
int count1(long a)
{int s=0;
 while (a)
 {s++;
  a&=(a-1);
 }
 return s;
}
long sum(long n)
{long s=0;
 for (long i=1;i<=n;i++) s+=count1(i);
 return s;
}
long sum1(long n)
{int k=0;
 long s,t=0;
 while (t<=n) {t=(t<<1)|1; k++;}
 t>>=1;
 k--;
 s=k*((long)1<<(k-1));
 for (long i=t+1;i<=n;i++) s+=count1(i);
 return s;
}
int main(void)
{long n;
 cin>>n;
// cout<<sum(n)<<endl;
 cout<<sum1(n)<<endl;
 return 0;
}
```

Round 1 / Task B3. ARRANGING FOR A PARTY

Starting with a man, we will realize a "greedy" linear strategy of arranging, which maximally uses women and minimally uses men. So we place women on the second and the third seat. While it is not possible to place another woman, the fourth seat will be occupied by a man. Alas, the next place cannot be for a woman either (this would result in a man with a woman on each side), so we are forced to use another man. This process continues while there are men and women: we place two women (one, if there are no two left), followed by two men (one, if he is the last one). We end up at one of these three possible stages:

- Nobody left at all. If the last seated is a man, the arrangement is good, otherwise it cannot be realized according the rules;
- One woman (or more) left. No way to arrange the table by the rules;
- One man (or more) left. We place them at the end without hindrance, obtaining a correct arrangement.

Actually, it appears that a correct arranging can be done when men's count is greater than women's, it cannot be done when it is less and in case of equality it can be done when this number is even. But the described strategy doesn't need these considerations. There is one case when it doesn't give correct answer: if there are one man and one woman they, obviously, are placed correct on the table. This case has to be considered separately.

Realization

```
#include <iostream>
using namespace std;
int M1,N1,M2,N2;
int arr(int M,int W,char *r)
{int i=0;
 r[i++]='1';M--;
 if (!M) {r[i++]='0';
          W--;
          if (!W) {r[i]=0; return 1;}
          return 0;
         }
 while (M>0 && W>0)
 {r[i++]='0';W--;
  if (W) {r[i++]='0';W--;}
  r[i++]='1';M--;
  if (M) {r[i++]='1';M--;}
 }
 if (W) return 0;
 while (M--) r[i++]='1';
 r[i]=0;
 return 1;
}
int main(void)
{char b[256];
 cin>>M1>>N1;
 cin>>M2>>N2;
 if (arr(M1,N1,b)) cout<<b<<endl; else cout<<"NO\n";
 if (arr(M2,N2,b)) cout<<b<<endl; else cout<<"NO\n";
 return 0;
}
```

Round 1 / Task C1. TRIANGLES

Let the lengths of the sides of the triangle be a, b and c. We can assume that $a \le b \le c$. Besides, for a triangle with lengths of the sides a, b and c to exist, the sum of any two of these numbers should be greater than the third number. As $a \le b \le c$, it is enough that $a + b > c$. Now, we have to generate all triples a, b and c, for which $a \le b \le c$, $a + b + c = P$ and $a + b > c$ and to count them.

```
#include<iostream>
using namespace std;

int main()
{
    int P, a, b, c, s = 0;
    cin >> P;
    for(int a=1; 3*a <= P; a++)
      for(int b=a; a + 2*b <= P; b++)
        { c = P - a - b;
          if (a + b > c) s++;
        }
    cout << s << endl;
    return 0;
}
```

Round 1 / Task C2. CLOCK

To solve the problem first we transform the two readings in minutes. If the second reading denotes an earlier time than the first one, we add 1440 minutes (= 24 hours). We obtain the wanted time in minutes by subtracting the first reading of the clock display from the second. Then the result should be written in a form of the input data.

```
#include<iostream>

using namespace std;
```

```
int main()

{ char a1, a2, a3, a4 , a5, b1, b2, b3, b4, b5;
  cin >> a1 >> a2 >> a3 >> a4 >> a5;
  cin >> b1 >> b2 >> b3 >> b4 >> b5;
  int a = (10 * (a1 - '0') + (a2 - '0'))*60 + 10*(a4 - '0') +
(a5 - '0');
  int b = (10 * (b1 - '0') + (b2 - '0'))*60 + 10*(b4 - '0') +
(b5 - '0');
  if (a >= b) b = b + 1440;
  int c = b - a;
  if (c/60 < 10) cout << 0;
  cout << c/60 << ':';
  if (c%60 < 10) cout << 0;
  cout << c%60 << endl;
  return 0;
}
```

Round 1 / Task C3. MINIMAL DIFFERENCE

The problem can be solved using an exhausting algorithm (the cases are 24 in total), generating, for example, all permutations of the given four digits, and creating form the first couple ***a***, and from the second couple ***b***, then calculating their difference ***a***-***b*** and memorizing the minimal positive difference. There exists another, more effective algorithm:

- As we are looking for the minimal difference, the most significant digits have to be most close one to another, so we can choose one of the couple of digits with minimal difference to be first digits.
- After selecting the first digits, there is only one reasonable choice for the second ones, to obtain a less difference: the bigger of them should be for the subtrahend, and the less one – for the minuend.
- During this process we memorize the least difference. There are at most three possible choices for the couple of most significant digits, which makes this algorithm more effective.

Sorting input data makes programming easier, no matter which approach is used.

Realization

```
#include <iostream>
using namespace std;
int a[4];
int MaxNo(int start)
{int i,m=start;
 for(i=start+1;i<4;i++) if (a[i]>a[m]) m=i;
 return m;
}
void selSort(void)
{int i,j,c;
 for (i=0;i<3;i++){j=MaxNo(i);
                   c=a[i];
                   a[i]=a[j];
                   a[j]=c;
```

```
                    }
}
int better(void)
{int i,j,m=100,p,q,d=10;
 selSort();
 for (i=0;i<3;i++) if (a[i]-a[i+1]<d) d=a[i]-a[i+1];
 for (i=0;i<3;i++)
  if (a[i]-a[i+1]==d)
  {p=10*a[i];
   q=10*a[i+1];
   switch(i)
   {case 0:{p+=a[3];q+=a[2];break;}
    case 1:{p+=a[3];q+=a[0];break;}
    case 2:{p+=a[1];q+=a[0];}
   }
   if (p-q<m) m=p-q;
  }
 return m;
}
int main(void)
{int i;
 for (i=0;i<4;i++) cin>>a[i];
 cout<<better()<<endl;
 return 0;
}
```

Round 1 / Task D1. RECTANGLES

Let the lengths of the sides of the rectangle be a and b. We can assume that $a \leq b$. S, a and b are integers. Therefore there exists a rectangle with side a and area S only if S is divisible by a (otherwise b cannot be an integer). To include only rectangles for which $a \leq b$ we give a only values for which $a^2 \leq S$.

```
#include<iostream>

using namespace std;

int main()
{ int S, a, sum = 0;
  cin >> S;
  for(int a=1; a*a <= S; a++)
    if (S % a == 0) sum++;
  cout << sum << endl;
  return 0;
}
```

Round 1 / Task D2. DIVISIBILITY BY 3

Sorting the digits in increasing order $a<b<c$ gives us the candidates for the result (increasingly): a, b, c, $\overline{ab}$, $\overline{ac}$, $\overline{bc}$ and $\overline{aaa}$. Of course, if at least one digit is divisible by 3, the smallest one is the one-digit solution. Otherwise, we check the two-digit numbers in increasing order (there is no point in considering other alternatives – if $\overline{ab}$ is not divisible by 3, neither is $\overline{ba}$, according to the divisibility rule; $\overline{aa}$ is divisible by 3 only if a is. Finally, a number with three equal digits is always divisible by three (according to the same rule) and $\overline{aaa}$ happens to be the smallest three-digit number to be collected using the given digits.

Realization

```
#include <iostream>
using namespace std;
int main(void)
{int a,b,c,d;
 cin>>a>>b>>c;
 if (a>b){d=a;a=b;b=d;}
 if (b>c){d=b;b=c;c=d;}
 if (a>b){d=a;a=b;b=d;}
 if (a%3==0) cout<<a;
 else if (b%3==0) cout<<b;
 else if (c%3==0) cout<<c;
 else if ((a+b)%3==0) cout<<a<<b;
 else if ((a+c)%3==0) cout<<a<<c;
 else if ((b+c)%3==0) cout<<b<<c;
 else cout<<a<<a<<a;
 cout<<endl;
 return 0;
}
```

Round 1 / Task E1. ANT

The length of the course that the ant traveled is equal to the double sum of all integers from 1 through n.

```
#include<iostream>

using namespace std;

int main()
{
    int n, s = 0;
    cin >> n;
    for(int i=1; i<=n; i++)
      s = s + i;
    s = 2*s;
    cout << s << endl;
    return 0;
}
```

Round 1 / Task E2. NUMBERS

Let us denote the missing digit by *i*. Then the number $\overline{a*b} = 100.a + 10.i + b$. The digit *i* can be 0, 1, 2, 3, 4, 5, 6, 7, 8 or 9. For any of these possibilities we check if the number $\overline{a*b} = 100.a + 10.i + b$ is divisible by *k*.

```
#include<iostream>
using namespace std;

int main()
{   int a, b, k, n, s=0;
    cin >> a >> b >> k;
    for(int i=0; i<=9; i++)
      { n = 100*a + 10*i + b;
        if (n%k ==0) s++;
      }
    cout << s << endl;
    return 0;
}
```

Round 2

Round 2 / Task A1. SUMS OF PRIME NUMBERS

To solve the task we have to take the following steps:

- find the primes smaller or equal to *n*;
- find the smallest prime *p* greater than *n*;
- find the remainder of the division of *S*(*n*) by *p*.

To find the primes smaller or equal to *n*, we can use the Sieve of Eratosthenes. We write the primes which we obtain in the array *a* and their number in the variable *cnt*. The function *eratosthenes*(*n*) returns a value 1 if *n* is a prime number and 0 in all the remaining cases.

Second, we check if every natural number greater than *n* is a prime, until we find *p*.

For the third step of the solution, we use one-dimensional array *s*. If $i = 1, 2, \ldots, n$ then *s*[*i*] is the remainder of the division of the number of ways in which *i* can be written as a sum of primes by *p*; if *i* is a prime then the sum can consist of only one number. *s*[*i*] can be obtained in *k* steps ($k = 0, 1, \ldots, cnt - 1$). At the k^{th} step *s*[*i*] is the remainder of the division of the number of ways in which *i* can be written as a sum of primes by *p* but the sum must be formed by the primes from *a*[0] to *a*[*k*]. If *n* is a prime number, *s*[*n*] should be changed to exclude the writing of *n* as a sum of one prime number.

```
#include <iostream>
using namespace std;

int const MAXN = 50010;

bool sieve[MAXN];
long a[MAXN];
long s[MAXN];
long cnt;

int eratosthenes(long n)
{ for(long i = 2; i <= n; i++)
    if (!sieve[i])
      { a[cnt] = i;
        cnt++;
        for(long j = i+i; j <= n; j+= i)
          sieve[j] = true;
      }
  if (a[cnt-1] == n) return 1;
  return 0;
}

int main()
{  long n;
   cin >> n;
   if (n == 1)
```

```
     { cout << 0 << endl;
       return 0;
     }

   int is_n_prime;
   is_n_prime = eratosthenes(n);
   long p = n+1;
   bool is_p_prime;
   do
     { is_p_prime = true;
       long i = 0;
       while(is_p_prime && a[i]*a[i] <= p)
         { if (p%a[i] == 0)
             {is_p_prime = false;
              p++;
             }
           i++;
         }
       }
   while(!is_p_prime);

   s[0] = 1;
   for(long k = 0; k < cnt; k++)
     { long x = a[k];
       for(long i = x; i <= n; i++)
         s[i] = (s[i]+s[i-x])%p;
     }

   s[n] = (s[n]+p-is_n_prime)%p;
   cout << s[n] << endl;

   return 0;
}
```

Round 2 / Task B1. DECIMAL FRACTION

Any common fraction can be written as a terminating or non-terminating periodic decimal fraction. According to the task, the terminating decimal fractions should be considered non-terminating periodic decimal fraction with a period (0). To solve the task, it is enough to find the number in digits of the period and the digits after the decimal point including the digits of the first period. Let us consider the sequence a_0, a_1, a_2, a_3, …, in which $a_0 = a$, and a_n is the remainder of the division of $10a_{n-1}$ by b. As the remainders of the division by b are limited in number, the members of the sequence will be repeated periodically. Let us denote by m and $m + t$ the numbers of the first pair of repeating members of the sequence for which $m \neq 0$. Let the sequence c_1, c_2, c_3, … consist of the first, second, third, … digit after the decimal point of the decimal fraction in consideration. It is easy to see that c_n is the quotient of the division of $10a_{n-1}$ by b. Then the period of the decimal fraction will consist of t digits and $c_{n+t} = c_n$ for any $n > m$. Therefore it is enough to find the first $m + t$ members of the sequence c_1, c_2, c_3, … .

To find m and t, we can use a one-dimensional array d. At the beginning, the values of the elements of the array are 0. After computing a_n, if $d[a_n] = 0$, then we give $d[a_n]$ a value n. Otherwise the sequence a_1, a_2, a_3, … already has an element with value a_n, its number is $d[a_n]$, therefore $m = d[a_n]$, $t = n - m$.

We find the p digits that we are looking for in the following way: if $n \leq m$, then the value of c_n has already been calculated; otherwise it is equal to the value of c_i, where i is the sum of $m + 1$ and the remainder of the division of $n - m - 1$ by t.

```
#include <iostream>
using namespace std;
const long MAXB = 30000010;
long d[MAXB];
short c[MAXB];

int main()

{ long a, b;
  long long k;
  int p;
  cin >> a >> b >> k >> p;
  long m, t;
  bool pr = true;
  long n = 0;
  while(pr)
  { n++;
    c[n] = (10*a)/b;
    a = (10*a)%b;
    if (d[a] == 0)
      d[a] = n;
    else
      { m = d[a];
        t = n - m;
        pr = false;
      }
  }
  for(int j=0; j<p; j++)
    if (k+j < m+1) cout << c[k+j];
    else
      { long x;
        x = (k+j-m-1)%t;
        cout << c[m+1+x];
      }
  cout << endl;
  return 0;
}
```

Round 3

Round 3 / Task A1 / B1. GRAPH DIAMETER

A solution that is based on the Floyd’s All-shortest paths algorithm, would get 50% of the scores.

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

const long INF = 1<<25;
int a[1024][1024];
int n;

int main()
{ scanf("%d",&n);

  for(int x=1; x<=n; x++)
  for(int y=1; y<=n; y++)
    a[x][y]=INF;

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

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

  for(int z = 1; z<=n; z++)
  {
    for(int x=1; x<=n; x++)
    for(int y=1; y<=n; y++)
      if(a[x][y] > a[x][z]+a[y][z])
        a[x][y] = a[x][z]+a[y][z];
  }

  int diam=0;
  for(int x=1; x<=n; x++)
  for(int y=1; y<=n; y++)
    if(diam < a[x][y]) diam = a[x][y];
  printf("%d\n",diam);

  return 0;
}
```

We may use the property that if an undirected graph G with n vertices and $n-1$ edges is connected then this graph is a tree.

Consider an arbitrary DFS-spanning tree of G. Let r be the root, and G_1, G_2, …, G_k be the subtrees of r with roots r_1, r_2, …, r_k.

Denote by $h(x)$ the height of the DFS-subtree with root x (the longest path from the root to a leaf), and denote by $c(x,y)$ be the length of the edge $\{x,y\}$.

The following formula holds:

$$\mathrm{diam}(G) = \max(\mathrm{diam}(G_i),\ c(r,r_i)+h(r_i) +\ c(r,r_j)+h(r_j) \mid i=1,2,\ldots,k,\ \ j=1,2,\ldots,k,\ \ i\neq j)$$

The meaning is that for the longest path with length equal to the diam(G) there are two possibilities:

1) The path is somewhere in a subtree G_i
2) The path is between two leaves belonging to the different subtrees G_i and G_j.

These considerations could be built in the general DFS algorithm.

```
#include <cstdio>
#include <vector>
using namespace std;

vector<int> a[1024];
int c[1024][1024];
long h[1024],d[1024];
int n;
long diam;

long height(int x, int z)
{ long res = 0;
  for(int i=0; i<a[x].size(); i++)
  { int y = a[x][i];
    if(y != z)
    { long hy = height(y,x);
      if(res < c[x][y] + hy) res = c[x][y] + hy;
    }
  }
  h[x] = res;
  return h[x];
}

void dfs(int x, int p)
{ long h1 = 0, h2 = 0;
  d[x] = 0;
  for(int i=0; i<a[x].size(); i++)
  { int y = a[x][i];
```

```
    if(y != p)
    { dfs(y,x);
      if(d[x] < d[y]) d[x] = d[y];
      int k = c[x][y] + h[y];
      if(k>h1) {h2 = h1; h1 = k; }
      else if(k>h2) h2 = k;
    }
  }
  h[x] = h1;
  if(d[x] < h1+h2) d[x] = h1 + h2;
  if(diam < d[x]) diam = d[x];
}

int main()
{ scanf("%d",&n);

  for(int i=1; i<n; i++)
  { int x, y, d;
    scanf("%d%d%d",&x,&y,&d);
    a[x].push_back(y);
    a[y].push_back(x);
    c[x][y] = c[y][x] = d;
  }

  dfs(1,0);

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

  return 0;
}
```

Round 3 / Task A3 / B3. COLORING

We can count all configurations we are interested in by generating them. For the purpose we can use a two-dimensional array a. The element $a[x][y]$ has a value 0 if the cell in row x and column y is colored in white and a value 1 when it is colored in black. We generate the elements of the array a row by row starting from the first and in each row the elements are generated from left to right. For our convenience we can assume that the table has two additional rows with numbers 0 and $n+1$ and two additional columns with 0 and $k+1$. We shall assume that the cells in these rows and columns are colored in white. In this way each cell from the original table will have a neighbouring cell to the left, to the right, at the top and at the bottom. There exist the following possibilities for coloring of a given cell:

- the cell can be colored both in black and in white;
- the cell can be colored only in black;
- the cell can be colored only in white;
- the cell can be colored neither in black nor in white.

We decide on the possible ways of coloring using the information about the top, left and right cells of the given cell. As the current cell is the last of the cells neighbouring of the top one, the choice of color guarantees the requirement that the top neighbouring cell have exactly one neighbouring cell, colored in black. Exceptions are the following situations:

- for the cells of the first row of the table the top and right neighbouring cells of the current one do not lead to any limitations in the choice of colouring;
- for the cells of the first column of the table the left neighbouring cell of the current one has no information about the way the current one is colored;
- no coloring for the cells of the $(n+1)^{\text{th}}$ row of the table is done because they are colored in white. However, they have to be visited during generation because passing through them we check if their neighbouring cells (the cells of the n^{th} row of the table) meet the requirement for a single black neighbouring cell.

```
#include <iostream>
using namespace std;

const int MAXN = 512;
const int MAXK = 35;

short a[MAXN][MAXK];
short n, k;
int br;

void Gen(short x, short y)
{ if (x == n+2)
     { br++;
       return;
     }
   short nextx, nexty;
   if (y < k) { nextx = x; nexty = y+1;}
   else {nextx = x+1; nexty = 1;}
   short l = 0;
   if (y != 1) l = a[x][y-2]+a[x-1][y-1];
   if (x == 1)
     { a[x][y] = 0; Gen(nextx, nexty);
       if (l == 0)
         {a[x][y] = 1; Gen(nextx, nexty);}
       return;
     }
   short t = a[x-1][y-1] + a[x-1][y+1] + a[x-2][y];
   if (x == n+1)
     { if (t == 1) Gen(nextx, nexty);
       return;
     }
   short r = a[x-1][y+1];
   if (t == 1) {a[x][y] = 0; Gen(nextx, nexty); return;}
   if (t == 0 && l == 0 && r == 0) {a[x][y] = 1; Gen(nextx,
nexty);}
}

int main()
{  cin >> n >> k;
```

```
    Gen(1,1);
    cout <<  br << endl;
    return 0;
}
```

Round 3 / Task A4 / B4. TABLE

Let's first consider the concept of equivalence defined in the problem. In each class of equivalence using row and/or column rearrangement we can achieve the following:

- Place the least of the twelve numbers in the upper left corner of the table;
- Sort ascending the numbers in the first row;
- Sort ascending the numbers in the first column.

It is clear that two equivalent tables turn to be identical in this "canonical" representation. And vice versa: if two tables are not identical in this representation, they are not equivalent. So we can create (and count) only "canonical" arrangements – the rest ones are equivalent to one of them.

One of the obvious necessary existence conditions for correct arrangement is divisibility by 6 of the sum of all numbers, as this sum is even when calculated by rows, and divisible by 3 when calculated by columns. In fact, the remainders of the numbers modulo 6 play in this problem an exceptional role. We can, for example, avoid working with long integers by replacing each of them with a less one with the same remainder, however, preserving order relationships between input data (on account of the canonic representation). So work data can actually contain numbers no greater than 71. The easiest way to do it is after sorting (ascending, for example) the input data. We shall follow our considerations supposing that input data is sorted this way (and replaced, eventually) in an integer array `data` with smallest index of 0 and greatest index of 11.

It is important afterwards to evaluate the biggest possible number of canonic tables.

data$_0$	r_0	r_1	r_2
c_0	rest$_0$	rest$_1$	rest$_2$
c_1	rest$_3$	rest$_4$	rest$_5$

The upper left corner is fixed. We have to select two from the rest 11 values (c_0 and c_1) to form the first column and three from the rest 9 (r_0, r_1 and r_2) to fill in the first row, thus creating a particular "signature" to every class of equivalence – $\{(c_0, c_1), (r_0, r_1, r_2)\}$. Selected elements are sorted, so selection order doesn't matter. This makes $\binom{11}{2}\binom{9}{3} = 4620$ possible selections. The rest 6 table cells can be filled in by the rest 6 numbers arbitrary, and their arrangements give rise of a new equivalence class each. That makes for each signature 6!=720 possibilities, or in sum at most 4620×720=3326400 possible classes of equivalence. If every input element can be put in every cell of the table (for example, this is obviously the case when all input numbers have the same remainder modulo 6), the obtained upper limit turns to be the final result, too.

A good organized backtracking algorithm can solve the problem in virtue of these considerations. If we manage to optimize it, it will be effective enough for the hardest cases. Here is a dynamic idea, applied to use already calculated results.

The ordered numbers in the signature modulo 6 form a "reduced signature": a five digit hexary number $\overline{c_0c_1r_0r_1r_2}$. The possibilities are 6^5=7776. The non-ordered remainders modulo 6 of the rest numbers (while each of them can take any place) identify the problem. If we choose to consider these remainders as total counts (b_0 – number of remainders 0, b_1 – number of remainders 1, …, b_5 – number of remainders 5), then $b_0+b_1+b_2+b_3+b_4+b_5=6$. However, even this rather exact estimation requires too much memory resource for directly applying the dynamic idea (over 13MB, which will not be effectively used), being hard to code in the sane time. It seems wiser to define a static array of pointers – one for each "reduced signature" pointing to dynamically designed list of already solved sub-problems with this "reduced signature". We can code the remainders count array ***b*** with "no economy" (but quickly) in 4 bytes.

It is possible, of course, to make a purely combinatorial realization on the same ideas.

Realization

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct Node {long hash;
             long value;
             struct Node *next;
            }Node;
typedef struct Node List;
int data[12],rem[6];
int used[12]={0};
int c[3][4];
int rowSum[3]={0},colSum[4]={0};
long count=0;
long cc,h;
int s;
List *L[7776]={NULL};
int rem6(char *s)
{int r=0;
 while (*s) r+=*s++;
 return (3*(*--s&1)+4*(r%3))%6;
}
int cmp(const void *a,const void *b)
{char *p=(char *)a,*q=(char *)b;
 int lp=strlen(p),lq=strlen(q);
 if (lp<lq) return -1;
 if (lp>lq) return 1;
 return strcmp(p,q);
}
void inp(void)
{char b[12][64];
 int i,r,t=0;
 for (i=0;i<12;i++) scanf("%s",b[i]);
 qsort(b,12,64,cmp);
 for (i=0;i<12;i++)
 {r=rem6(b[i]);
  rem[r]++;
  data[i]=6*t+r;
  if (i && data[i]<=data[i-1]){data[i]+=6;t++;}
 }
}
void init(void)
{used[0]=1;
```

```
 rem[data[0]]--;
 c[0][0]=rowSum[0]=colSum[0]=data[0];
}
void addToList(long h,long v,List **l)
{List *r=(List *)malloc(sizeof(Node));
 r->hash=h;
 r->value=v;
 r->next=*l;
 *l=r;
}
long inList(long h,List *l)
{while (l && l->hash!=h) l=l->next;
 if (l) return l->value;
 return -1;
}
void FreeList(List **l)
{if (*l) FreeList(&(*l)->next);
 free(*l);
 *l=NULL;
}
int makeSign(void)
{int r=c[1][0]%6,i;
 r=6*r+(c[2][0]%6);
 for (i=1;i<4;i++) r=6*r+(c[0][i]%6);
 return r;
}
long makeHash(int *b)
{long r=0;
 int i;
 for (i=0;i<6;i++) r=(r<<4)|b[i];
 return r;
}
void bt(int row,int col)
{int p;
 long r;
 if (row&&col>3) {cc++;
                  return;
                 }
 for (p=1;p<12;p++)
  if (!used[p])
  {if (col && !row && data[p]<c[row][col-1]) continue;
   if (row && !col && data[p]<c[row-1][col]) continue;
   if (col==3 && (rowSum[row]+data[p])&1) continue;
   if (row==2 && (colSum[col]+data[p])%3) continue;
   c[row][col]=data[p];
   rem[data[p]%6]--;
   rowSum[row]+=data[p];
   colSum[col]+=data[p];
   used[p]=1;
   if (!col && row<2) bt(row+1,col);
   else if (!col && row==2) bt(0,1);
```

```
    else if (!row && col<3) bt(0,col+1);
    else if (!row&&col==3)
    {h=makeHash(rem);
     s=makeSign();
     r=inList(h,L[s]);
     if (r>=0) count+=r;
     else
     {cc=0;
      bt(1,1);
      addToList(h,cc,&L[s]);
      count+=cc;
     }
    }
    else if(row<2) bt(row+1,col);
    else bt(1,col+1);
    rem[data[p]%6]++;
    rowSum[row]-=data[p];
    colSum[col]-=data[p];
    used[p]=0;
   }
}
int main(void)
{int i,lin;
 for (lin=0;lin<2;lin++)
 {inp();
  count=0;
  init();
  bt(1,0);
  printf("%ld\n",count);
  for (i=0;i<7776;i++) FreeList(&L[i]);
 }
 return 0;
}
```

Round 3 / Taѕк A5 / B5. MATRYOSHKA DOLLS

Analysis

We can give each room in the labyrinth coordinates *(x, y)*, *x* being the row of the room and *y* – the column (we count from *1* to *N* from top to bottom and from left to right). Let M_{xy} be the size of the doll in room *(x, y)*. We can define path in the labyrinth as a string of rooms, such that each one, except the first, contains a larger toy than the previous and its coordinates are not smaller than the coordinates of the previous. Obviously, Peter the Hacker can go through the labyrinth passing through all the rooms from a single path and collecting all the dolls in them. This way, we will know the answer of the problem if we can find the length of the longest path.

Solution

We will solve this problem using dynamic programming. Lets define a sub problem $f(i, j)$, which finds the length of the longest path that ends in room (i, j). We can calculate the value of $f(i, j)$ recursively the following way: $f(i, j)=1+max\{f(k, l) \mid k\leq i, l\leq j, (k, l)\neq(i, j), M_{kl}<M_{ij}\}$.

Direct implementation would be to use a square matrix containing the values of $f(i, j)$. To calculate each value we can cycle through all the sub problems according to the above formula and find the maximum. This will lead us to overall time complexity of $O(N^4)$. Of course, it will not work on all the tests with the given restrictions and will receive 50 points.

We can optimize the calculations for each sub problem by using a data structure with logarithmic complexity for update and query (for example binary indexed tree) for finding the maximal values. Let us take another look to the recursive formula. Obviously, the current longest path cannot have a negative length, so the maximum we are looking for is at least zero. Therefore, if we initialize the structure with zeroes, the condition $(k, l)\neq(i, j)$ becomes redundant. There are three more conditions left which leads us to a three-dimensional indexed tree. Such a solution is quite hard to implement and it has time complexity of $O(N^2.ln^3 N)$ and memory complexity of $O(N^4)$.

To build a better solution we need to loose one more condition and its corresponding dimension in the tree. Obviously, every sub problem is influenced only by sub problems which corresponding rooms contain smaller dolls. Therefore, we can calculate the values in ascending order of their related toy sizes (the size is unique). This way the condition $M_{kl}<M_{ij}$ becomes redundant, too, because in the tree there will be no updates for sub problems with smaller toys than the toys of the one we are currently calculating. A two-dimensional indexed tree leads to a solution with complexity $O(N^2.ln^2 N)$, which will receive 100 points.

Scoring

If the solution is implemented well enough it will receive points according to its complexity: *brute force* – 10; $O(N^4)$ – 50; $O(N^2.ln N)$ – 100.

Round 3 / Task C1. GAME

We will use two one-dimensional arrays:

array *a* in which *a*[*i*]=0, when the *i*-th cell is white, and *a*[*i*]=1 when it is black;
array *d* in which *d*[*i*] is the distance (the minimum number of moves needed for moving the pawn from the initial cell to the current cell); *d*[*i*] = –1, when the *i*-th cell is not accessible.

We start with *d*[*x*]=0 and *d*[*i*] = –1 for the remaining cells. Now we apply the breadth first search algorithm. For the current cell *z* we scan in the four directions (up, down, left, right) for unvisited white cells. If such a cell is found, its distance should to be updated and the number of the cell inserted into the queue.

```
#include <cstdio>
#include <queue>
using namespace std;

int a[1000000];
int d[1000000];
int n;

queue<int> q;
```

```
int getRow(int z)
{ return 1+(z-1)/n; }

int main()
{ int x,y,b;
  scanf("%d%d%d",&n,&x,&y);

  scanf("%d",&b);
  for(int i=1; i<=b; i++)
  { int z;
    scanf("%d",&z);
    a[z] = 1;
  }

  for(int i=1; i<=n*n; i++)
    d[i] = -1;

  d[x] = 0;
  q.push(x);

  while(!q.empty())
  { int z = q.front();
    q.pop();

    int p;
    p = z-1;
    while(getRow(p)==getRow(z) && a[p]==0)
    { if(d[p]==-1) { d[p]=d[z]+1; q.push(p); }
      p--;
    }

    p = z+1;
    while(getRow(p)==getRow(z) && a[p]==0)
    { if(d[p]==-1) { d[p]=d[z]+1; q.push(p); }
      p++;
    }

    p = z-n;
    while(p>=0 && a[p]==0)
    { if(d[p]==-1) { d[p]=d[z]+1; q.push(p); }
      p = p-n;
    }

    p = z+n;
    while(p<=n*n && a[p]==0)
    { if(d[p]==-1) { d[p]=d[z]+1; q.push(p); }
      p = p+n;
    }
  }
  printf("%d\n",d[y]);

  return 0;
}
```

Round 3 / Task C2. TRANSFORMATIONS

Let a have n digits. Then there are $n-k+1$ groups of successive digits in a which can be changed at one move. Let us assign numbers to these groups: the first group will be the one containing the digits from the first to the k^{th} digit of a, the second group – containing the digits from the second to the $(k+1)^{\text{th}}$ digit of a, …, the last $(n-k+1)^{\text{th}}$ group – containing the digits from the $(n-k+1)^{\text{th}}$ to the n^{th} digit of a. To calculate the minimum number of moves with which b can be derived from a, we can do the following: we transform only the first group of digits with a minimum number of moves until the first digits of a and b are identical (the number of moves can be 0, 1, 2, 3, 4, 5, 6, 7 or 8). Then we do the same with the new number, using only the second group of the digits so that its second digit is identical to the second digit of b. We continue in the same way until the first $n-k+1$ digits of the number we transformed and the first $n-k+1$ digits of b are identical. If at this stage the transformed number and b are identical, the wanted minimum number of moves is the number of moves we have performed. If the two numbers are not identical, b can not be derived from a irrespective of the number of moves.

Now we have to prove that the algorithm we described leads to a correct result. Any sequence of m number of moves can be considered a sequence of m numbers each of which is between 1 and $(n-k+1)$ and shows which group was used for the move. For example, if a = 12349 and k = 2, the sequence 2, 1, 3, 1 means that we have successively derived the numbers: 13449, 24449, 24559 and 35559. One can easily see that if we exchange the positions of some of the numbers, the final result will not be changed (for example, the sequence 1, 1, 2, 3 generates the numbers 23349, 34349, 35449 and 35559). Therefore we can make all necessary moves first with the first group, then with the second etc.

```
#include<iostream>
#include<string>
using namespace std;

int main()
{
    string a,b;
    int k, n, p, br = 0;
    cin >> k >> a >> b;
    n = a.size();
    for(int i=0; i < n-k+1; i++)
    { p = (9 + b[i] - a[i])%9;
      br += p;
      for(int j=0; j<k; j++)
        a[i+j] = '1' + (a[i+j]-'1'+p)%9;
    }
    if (a == b) cout << br << endl;
    else cout << 0 << endl;
    return 0;
}
```

Round 3 / Task C3. STRANGE WORDS

The solution of this task is based on organizing a cycle which reads the text word by word till the input reaches the symbol for the end of the text. For each read word `s`, the program should check if it is strange and if it is so – calculates its length `d` and compares it to the length of the longest word found till now – `md`. If the found word has greater length – it becomes the longest strange word – `ms=s`.
The check for strangeness is made by the function sword, which uses a stack of chars. The word is scanned from left to right and the following things are done for each character from it:

– If the stack is not empty, and the character written on its top matches the current one, it is erased from the stack.

– In the other case, the current character is written on the top of the stack.

As a secondary effect the function transforms the word by replacing all the capital letters into small ones.
Let's notice that the function gets as a second parameter the length of the word, so that the reapeat calculations is avoided.
For the solution we use the standard types `string` and `stack`.

```
#include<string>
#include<iostream>
#include<stack>
using namespace std;
int sword(string &s,int n)
{
    stack<char> st;
    int i,l, d='a'-'A';
    char ch;
    for(i=0;i<n;i++)
    {
        if(s[i]>='A'&&s[i]<='Z')s[i]+=d;
        l=0;
        if(!st.empty())
        {
            ch=st.top();
            if(s[i]==ch){st.pop();l=1;}
        }
        if(!l) st.push(s[i]);
    }
    return (st.empty());
}
int main()
{
    string s, ms;
    int md =0,d ;
    while(cin>>s)
    {
        d=s.size();
        if(sword(s,d))
            if(d>md)
            {
                md=d;
                ms=s;
            }
    }
    cout<<ms<<endl;
}
```

Round 3 / Task C4. POLYGON

The area of the polygon is calculated by adding or subtracting the areas of rectangles formed by the horizontal sides and the *x*-axis. The area of a given rectangle is added, if a vertical line from point (*x* + 0.5*, y* + 0.5), where *x* and *y* are the coordinates of the corresponding side left end, crosses an odd number of the other polygon sides. If the number of intersections is even, the area is subtracted.

```
#include <iostream>
using namespace std;
long n, x1[1000], x2[1000], y[1000], s=0;
int cross(double a, long b, long c)
{
    return b<a && a<c;
}
int inside(int k)
{
    double a=x1[k]+0.5, d=y[k]+0.5; int c=0;
    for (int i=0;i<n;i++)
    {
        if (y[i]>d)
           if (cross (a, x1[i], x2[i]))c++;
    }
    return c%2;
}
int main()
{
    double x;
    cin>>n;
    for (int i=0;i<n;i++)
    {
        cin>>x1[i]>>x2[i]>>y[i];
    }
    for (int i=0;i<n;i++)
    {
          long st=y[i]*(x2[i]-x1[i]);
          if (inside(i))s-=st;
          else s+=st;
     }
     cout<<abs(s)<<endl;
     return 1;
}
```

Round 3 / Task C5. SCALE

The integer *m* is presented as a sum $d[0]*3^0 + d[1]*3^1 + d[2]*3^2 + d[3]*3^3 + \ldots + d[k]3^k$, where the coefficients *d*[i] have values –1, 0 or 1. The object with weight *m* and the masses preceded by a coefficient –1 are put on the left hand scalepan, while the masses preceded by a coefficient 1 are put on the right hand scalepan. The masses preceded by a coefficient 0 are not used.

```
#include <iostream>
using namespace std;
int main()
{
     long n, i=0, d[20];
     long m, x;
     cin>>n>>m;
     x=m;
     while (x>0)
     {
            d[i]=x%3; if (d[i]==2)d[i]=-1;
            x=(x-d[i])/3; i++;
     }
     cout<<m;
     x=1;
     for (int i=0;i<n;i++)
     {
            if (d[i]==-1)cout<<" "<<x;
            x*=3;
     }
     cout<<endl;
     x=1;
     for (int i=0;i<n;i++)
     {
            if (d[i]==1)
               if (i<n-1)cout<<x<<" ";
               else cout<<x;
            x*=3;
     }
     cout<<endl;
     return 1;
}
```

Round 3 / Task C6. DANCE

Obviously, if *x* or *y* are odd numbers the result of the program is 0 because the dancer will either not reach or pass by his partner. Since one dance step consists of two foot steps, we may change *x* and *y* to *x*/2 and *y*/2.

Some participants could model this task to another one, where the aim is to calculate the number of movements on a grid from the bottom-left to the top-right corner with limited movements only to the right or up. This number is equal to the binomial coefficient $C(x+y, x)$. However, calculating this using a recursion will be a mistake because of the time limit.

So, the whole number of ways to reach the spot from the initial position of the dancer (i, j) is the sum of the number of ways from position $(i–1, j)$ and those from position $(i, j–1)$.

Using dynamic programming technique, this number could be calculated using one two dimensional array or even one dimensional array, where every new number comes from the number being previously saved in place *j* plus the number saved in place *j*–1.

```
#include<iostream>
using namespace std;

int main()
{
int x, y, i, j, temp ;
long long t[101];

cin>>x>>y;
if (x>y) {temp=x; x=y; y=temp;}

if (x%2 || y%2) { cout<<0<<endl; return 0;}

x/=2;
y/=2;
for (i=0; i<=y; i++)
    t[i]=1;
for (i=1; i<=y; i++)
    for (j=1; j<=x; j++)
        t[j]=t[j-1]+t[j];

cout<<t[x]<<endl;

return 0;
}
```

Round 3 / Task D2. TREASURE

We consider all given *N* lines as elements of an array of strings. In every string we find the smallest character. In the program, the function **`char`** `search(string s,` **`int`** `m)` returns the smallest character, which does not occur in `s` and which is greater than the smallest character `m` in the same string, or returns `'.'` in case the string `s` contains all letters from the alphabet.

To determine if a given character belongs to the string, we use an auxiliary array `b` with 123 elements, because all considered characters are letters and the greatest one of them by the alphabetical order is ‘z’, and it has an ASCII code 122. Starting with all zeros, the values of the elements of `b` is either 0, when the corresponding character does not belong to the string, or 1, otherwise. We do not consider characters with ASCII codes between the sets of all capital and all small letters, assigning 1’s to their positions. After having filled all the positions in `b`, we can find the desired character by searching for the first element in `b` which is placed after the position of `m` and which is equal to 0.

```
#include <iostream>
using namespace std;
int b[123];
string a[10000];

char search(string s,int m)
{
 int k=0;
 for(int i=0;i<=122;i++)
 b[i]=0;
 for(char ch='Z'+1;ch<'a';ch++) b[ch]=1;
 while(k<s.length()) {b[s[k]]=1;k++;}
 for(int i=m;i<=122;i++)
 {
 if(b[i]==0) return (char)i;
 }
return '.';
}

int main()
{ int n;
 int min;
     cin>>n;
     for(int i=0;i<n;i++)
     cin>>a[i];
     for(int i=0;i<n;i++)
     {  min=200;
          for(int j=0;j<a[i].length();j++)
          {if(min>a[i][j]) min=a[i][j];}
        cout<<search(a[i],min);
     }
cout<<endl;
}
```

Round 3 / Task D3. DIGIT

We are modeling the process using the string stream tool in C++. In the first loop, we put all integers from 1 to 3000 into the string stream buffer *a*. Then we get *n* characters, one by one from the buffer.

```
#include <iostream>
#include <sstream>
using namespace std;

int main()
{ int n;
  cin >> n;

  stringstream a;
  for(int x=1; x<=3000; x++)
    a << x;

  char ch;
  for(int i=1; i<=n; i++)
    a >> ch;

  cout << ch << endl;

  return 0;
}
```

Round 3 / Task E1. RHOMBS

The program reads integer N that shows how many inscribed rhombs are formed the figure. The number of printed rows is 2N+1. Each row contains two parts: definite number spaces (sp) and definite number (br) *. The first row consists of N spaces and one asterisk.

1. Through the first cycle **for,** we draw the first N rows, such that the number of spaces decreases by one for every next row, and the number of astersisks increases by 2.
2. Follow the row with N asterisks one space and N asterisks are added:

   ```
   for (i=1; i<=2; i++)  {
       for(j=1; j<=n;j++) cout <<'*';
       if (i==1) cout <<' '; }
   cout<<endl;
   ```
3. Through the next cycle **for,** we draw the next N rows, such that the number of spaces increases by one for every next row, and the number of asterisks decreases by 2.

```
#include <iostream.h>
using namespace std;
int main()
  { int i,j,n,sp,br=1;
    cin >>n;
    sp=n;
    for (i=1;i<=n;i++)
```

```
    { for (j=1;j<=sp;j++)
        cout <<' ';
      for (j=1;j<=br;j++)
        cout <<'*';
      cout <<endl;
      sp--;
      br+=2;     }
  for (i=1; i<=2; i++) {
    for(j=1; j<=n;j++)
       cout <<'*';
    if (i==1) cout <<' '; }
   cout<<endl;
   sp=1;
   br=2*n-1;
   for (i=1;i<=n;i++)
    {
      for (j=1;j<=sp;j++)
        cout <<' ';
      for (j=1;j<=br;j++)
        cout <<'*';
    cout << endl;
    sp++;
    br-=2;
    }
}
```

Round 3 / Task E3. IRREDUCIBLE FRACTION

Due to the bounds $0 < a < b < 1000$ it follows that any common denominator d is less or equal to 500. So, we try all possibilities for d from 2 to 500.

```
#include <iostream>
using namespace std;

int main()
{ int a,b,d;

  cin >> a >> b;

  for(d = 2; d<500; d++)
    while(a%d == 0 && b%d == 0)
    { a = a / d; b = b / d; }

  cout << a << " " << b << endl;

  return 0;
}
```