24. НАЦИОНАЛНА ОЛИМПИАДА ПО ИФОРМАТИКА

Решения на избрани задачи

2008

Кръг 1

Кръг 1 / Задача А1. ДРОБИ

Дробите, с които ще работим в задачата, може да разглеждаме като елементи на структура с два члена *p* и *q*, отразяващи съответно числителя и знаменателя на дробта. За да решим задачата, генерираме всяка дроб $\frac{a}{b}$, за която $k \le a < b \le n$, проверяваме дали е несъкратима и ако това е така, я добавяме към текущата променлива *s*, която използваме за намиране на търсената сума. За начални стойности на *s.p* и *s.q* задаваме съответно 0 и 1. След всяко сумиране на две дроби извършваме възможните съкращения в резултата.

```
#include<iostream>

using namespace std;

struct Fract
{ long long p,q; };

int nod(long long a, long long b)
{ long long 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;
           long long p = s.p*x.q + s.q*x.p;
           long long 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;
}
```

Кръг 1 / Задача А3. ПРИНАДЛЕЖНОСТ

Ограниченията допускат пряка проверка по правилата за всяко от входните числа – 18 десетични цифри се събират в 8-байтов целочислен тип. Такова решение рискува да е по-бавно за някои големи числа (**`inM1`** в реализацията).
От условието обаче следват следните свойства на двоичния запис на всеки от членовете на ***M***:

- Двоичните цифри са нечетен брой. Наистина, 1 е с една цифра; правило 2 добавя нови две цифри и следователно не сменя четността, а правило 3 обединява две предишни дължини, като добавя още една цифра. Тъй като в началото цифрата е една (нечетен брой), няма правило, по което четността да се смени.
- Правилата не пораждат числа, в които има съседни нули. Нещо повече – нулите могат да са само на четно място, съгласно правило 2.
- Всички числа с нечетна дължина в двоичен запис и нули (ако се срещат такива) на четни места участват в ***M***. Наистина – за дължина 1 това е тривиално вярно (по правило 1). Да допуснем, че е вярно за всички нечетни дължини, по-малки и равни на нечетното число ***n***. Да разгледаме сега числата с дължина на двоичния запис ***n***+2. Ако в този запис няма нула, той се поражда от числото 1 (правило 1) чрез (***n***+1)/2 пъти прилагане на правило 2. Ако има (поне) една нула (на нечетно място, разбира се), от лявата и от дясната й страна има записи с нечетни дължини и нули (ако се срещат) на нечетни места. Съгласно индуктивното предположение, там могат да се срещат всевъзможните записи със съответни дължини (не по-големи от ***n***). По правило 3 пък всеки такъв запис е от ***M***.

Използването на тези факти води до линеен по броя на двоичните цифри (т. е. логаритмичен по отношение на кандидата) алгоритъм за установяване на принадлежността към ***M***. (функцията **`inM2`**).

Реализация

```
#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;
}
```

Кръг 1 / Задача B1. МАТЕМАТИКА

Записаните на дъската n числа може да разглеждаме като елементи на едномерен масив a. Максималната сума може да се постигне, когато знаците за изваждане се поставят пред най-малките елементи на масива. Тъй като пред първото число знак не може да се поставя, то остава да се намерят най-малките k на брой числа измежду всички числа без първото. За решаване на задачата може да извършим следното:

1. Сортираме елементите на масива без $a[0]$ в низходящ ред.
2. Сумираме елементите на масива с номера от 0 до $n-k-1$.
3. От получената сума изваждаме останалите k елемента на масива.

```
#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;
}
```

Кръг 1 / Задача В2. СУМИ ОТ ЦИФРИ

Линейното сканиране (функция **sum**) работи достатъчно бързо за зададените ограничения, ако е написано добре. Сумата на цифрите в двоичния запис всъщност е броят на единиците. Можем да използваме известния алгоритъм с побитово AND за намирането на този брой (функция **count1** в реализацията).
За по-големи ограничения може да се използва следното подобрение (**sum1**):
Лесно се съобразява, че ако n е с едно по-малко от степен на двойката, т.е. $n=2^k-1$, двоичният му запис ще се състои само от единици (k на брой), а заедно с числата преди това записите все едно отразяват всички подмножества на едно k-елементно множество. Следователно, единиците ще се срещат по 2^{k-1} пъти на всяко от k места, значи търсената сума до такова n ще е $k.2^{k-1}$. Идеята може и да се доразвие, но в реализацията е използвана дотук: намираме най-голямото цяло число от вида 2^k-1 (т. е., само от единици в двоичния запис), ненадминаващо n, изчисляваме по горната формула сумата дотам и пускаме сканиране до n (функция **sum1**).

Реализация

```
#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;
}
```

Кръг 1 / Задача В3. ПОДРЕДБА ЗА ПАРТИ

Започвайки от мъж, ще реализираме "лакома" линейна стратегия на подреждане, максимално изразходваща жени и минимално мъже. Така на второ и трето място поставяме жени. Тъй като не можем да разположим следваща жена, на четвърто място поставяме мъж, след него, уви, още не можем да разположим жена (защото ще се получи мъж с две съседки), налага се да "използваме" още един мъж. Този процес продължаваме, докато още има за разполагане хора и от двата пола: две жени (една, ако е останала последната), следвани от двама мъже (един, ако е последният). След приключването на някой от половете можем да имаме следните ситуации:

- не са останали хора изобщо. Ако последният разположен е бил мъж, наредбата е добра, иначе – не може да бъде осъществена по правилата;
- останала е поне една жена – наредбата не може да се осъществи по правилата;
- останали са мъже (поне един) – спокойно ги разполагаме последователно накрая и получаваме една правилна наредба.

Всъщност, оказва се, че подредбата може да бъде изпълнена, ако броят на мъжете е по-голям от този на жените, не може, ако е по-малък, а ако са равни, може, когато броят на мъжете е четно число. Описаната стратегия обаче не се нуждае от тези разсъждения. Има един случай, при който тя не дава верен отговор – ако имаме само един мъж и една жена, те могат да се разположат на масата. Този случай трябва да се съобрази.

Реализация

```
#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;
}
```

Кръг 1 / Задача С1. ТРИЪГЪЛНИЦИ

Нека дължините на страните на триъгълника са a, b и c. Без ограничение може да считаме, че $a \le b \le c$. Освен това, за да съществува триъгълник с дължини на страните a, b и c, трябва сборът на всеки две от тези числа да е по-голям от третото число. Тъй като $a \le b \le c$, то достатъчно е само да бъде изпълнено условието $a + b > c$. Остава само да генерираме всички тройки числа a, b и c, за които $a \le b \le c$, $a + b + c = P$ и $a + b > c$ и да ги изброим.

```
#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;
}
```

Кръг 1 / Задача С2. ЧАСОВНИК

За да решим задачата първо преобразуваме началното и крайното показание на часовника в минути. Ако крайното показание на часовника показва по-малък час от началното, към него прибавяме 1440 (1440 минути са 24 часа). Търсеното време в минути получаваме като от крайното показание на часовника извадим началното. Остава резултатът да се изведе във формата на входните данни.

```
#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;
}
```

Кръг 1 / Задача С3. МИНИМАЛНА РАЗЛИКА

Задачата може да се реши с изчерпване на случаите (те са 24) като, например, се разгледат всички пермутации на дадените четири цифри, от първите две в получения ред се състави ***a***, а от вторите две – ***b***, образува се разликата им ***a-b*** и се запомни минималната положителна разлика по време на целия процес. Има и по-ефективен алгоритъм:

- Щом търсим минимална разлика, най-старшите цифри трябва да са възможно най-близки, значи ще изберем за старши цифри някои с най-малка разлика.
- След избора на първи цифри за умаляемото и умалителя, останалите две цифри имат единствено разумно разположение с цел минимизиране на разликата – по-голямата от тях да е втора цифра на умалителя, а по-малката – втора цифра на умаляемото.
- Избираме най-малката разлика при този процес. Може да има най-много три избора за двойка старши цифри, затова алгоритъмът е по-ефективен.

Разбира се, и за двата подхода подреждането на входните данни облекчава програмирането.

Реализация

```
#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;
}
```

Кръг 1 / Задача D1. ПРАВОЪГЪЛНИЦИ

Нека дължините на страните на правоъгълника са a и b. Без ограничение може да считаме, че $a \le b$. S, a и b са цели числа. Следователно правоъгълник със страна a и лице S съществува само, ако S се дели на a (в противен случай b няма да бъде цяло число). За да изброим само правоъгълниците, за които $a \le b$, на a даваме само такива стойности, за които $a^2 \le S$.

```
#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;
}
```

Кръг 1 / Задача D2. ДЕЛИМОСТ НА 3

Ако наредим цифрите по големина $a<b<c$, то кандидатите за решение (в нарастващ ред) са: a, b, c, $\overline{ab}$, $\overline{ac}$, $\overline{bc}$ и $\overline{aaa}$. Наистина, ако поне една цифра се дели на три, най-малката такава е едноцифреното решение. Иначе пробваме двуцифрените по големина (няма смисъл да се разглеждат другите варианти – ако $\overline{ab}$ не е кратно на 3, $\overline{ba}$ също не е –

според признака за делимост; $\overline{aa}$ е кратно на 3 само ако и самото a е такова). И накрая, число с три еднакви цифри винаги се дели на 3 (пак по признака за делимост), а $\overline{aaa}$ е и най-малкото трицифрено число, съставимо от трите цифри.

Реализация

```
#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;
}
```

Кръг 1 / Задача Е1. МРАВКА

Изминатият от мравката път е равен на удвоения сбор на естествените числа от 1 до 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;
}
```

Кръг 1 / Задача Е2. ЧИСЛА

Да означим липсващата цифра с i. Тогава числото $\overline{a*b} = 100.a + 10.i + b$. Цифрата i може да бъде 0, 1, 2, 3, 4, 5, 6, 7, 8 или 9. За всяка от тези десет възможности проверяваме дали числото $\overline{a*b} = 100.a + 10.i + b$ се дели на 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;
}
```

Кръг 2

Кръг 2 / Задача А1. СУМИ ОТ ПРОСТИ ЧИСЛА

Решаването на задачата преминава следните стъпки:

- Намиране на простите числа, които са по-малки или равни на n;
- Намиране на най-малкото просто число p, което е по-голямо от n;
- Намиране на остатъка от делението на $S(n)$ с p.

Намирането на простите числа, които са по-малки или равни на n може да се извърши чрез използване на решетото на Ератостен. В масива a записваме получените прости числа, а броя им – в променливата cnt. Функцията $eratosten(n)$ връща стойност 1, ако числото n е просто и 0 във всички останали случаи.

Втората стъпка извършваме като за всяко естествено число, по-голямо от n, проверяваме дали е просто, до намирането на p.

За реализацията на третата стъпка от решението използваме едномерен масив s. В него за $i = 1, 2, \ldots, n$ намираме остатъка при делене на p на броя на начините, по които числото i може да се представи като сума от прости събираеми, като в тази сума може да участва и само едно число (ако i е просто число). Намирането на $s[i]$ извършваме на k стъпки ($k = 0, 1, \ldots, cnt - 1$). В k-тата стъпка в $s[i]$ получаваме остатъка при делене на p на броя на начините, по които числото i може да се представи като сума от прости числа, като в тази сума могат да се използват само простите числа от $a[0]$ до $a[k]$. Ако числото n е просто, получения резултат за $s[n]$ трябва да се промени, за да се изключи представянето на n като сума от едно просто събираемо.

```
#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;
}
```

Кръг 2 / Задача B1. ДЕСЕТИЧНА ДРОБ

Всяка обикновена дроб може да се представи като крайна десетична или безкрайна периодична десетична дроб. Според условието на задачата крайните десетични дроби ще разглеждаме също като безкрайни периодични дроби с период (0). За да решим задачата, е достатъчно да определим броя на цифрите в периодичната част на числото и цифрите след десетичната запетая до приключването на първия период. Да разгледаме редицата a_0, a_1, a_2, a_3, … , в която $a_0 = a$, а a_n е остатъкът при делението на $10a_{n-1}$ с b. Тъй като остатъците при деление на b са краен брой, то членовете на редицата ще започнат периодично да се повтарят. Да означим с m и $m + t$ номерата на първата двойка повтарящи се членове на редицата, за които $m \neq 0$. Нека редицата c_1, c_2, c_3, … се състои от първата, втората, третата, … цифра след десетичната запетая на разглежданата десетична дроб. Лесно се съобразява, че c_n е частното при делението на$10a_{n-1}$ с b. Тогава периодът на десетичната дроб ще се състои от t цифри и ще е вярно,

че $c_{n+t} = c_n$ за всяко $n > m$. Следователно, достатъчно е да намерим само първите $m + t$ члена на редицата $c_1, c_2, c_3, \ldots$.
За намирането на m и t може да използваме едномерен масив d. В началото стойностите на елементите на масива са 0. След пресмятането на a_n, ако $d[a_n] = 0$, то на $d[a_n]$ присвояваме стойност n. В противен случай в редицата $a_1, a_2, a_3, \ldots$ вече има елемент със стойност a_n, неговият номер е $d[a_n]$ и следователно $m = d[a_n]$, $t = n - m$.
Интересуващите ни p цифри на числото определяме по следния начин: ако $n \leq m$, то стойността на c_n вече е пресметната, в противен случай тя е равна на стойността на c_i, където i е сборът на $m + 1$ и остатъкът от делението на $n - m - 1$ с 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;
}
```

Кръг 3

Кръг 3 / Задача A1 / B1. ДИАМЕТЪР НА ГРАФ

Решение базирано на алгоритъма на Флойд за намиране на всички най-къси пътища ще получи 50% от точките.

```
#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;
}
```

Може да се възползваме от свойството, че ако неориентиран граф G с n върха и $n-1$ ребра е свързан, то графът е дърво.
Да разгледаме произволно DFS-покриващо дърво на G. Нека r да е корена, а G_1, G_2, …, G_k са поддървета на r с корени r_1, r_2, …, r_k.
Да означим с $h(x)$ височината на DFS-поддърво с корен x (т.е. дължината на най-дългия път от корена до листо). Нека означим с $c(x,y)$ дължината на реброто $\{x,y\}$.

В сила е следната формула:

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

Смисълът на формулата е, че за най-дългия път с дължина равна на diam(G) има две възможности:
1) Пътят е някъде в поддърво G_i
2) Пътят е между две листа, принадлежащи на различни поддървета G_i и G_j.

Тези съображения могат да бъдат вградени в общия алгоритъм за търсене в дълбочина.

```
#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;
}
```

Кръг 3 / Задача А3 / В3. **ОЦВЕТЯВАНЕ**

Преброяването на всички интересуващи ни конфигурации може да извършим чрез генерирането им. За целта може да използваме двумерен масив a, като елементът $a[x][y]$ ще има стойност 0, ако клетката от таблицата, разположена в ред с номер x и стълб с номер y е оцветена в бяло и стойност 1, ако е оцветена в черно. Генерирането на елементите на масива a извършваме ред по ред, започвайки от първия, като във всеки ред елементите генерираме отляво надясно. За удобство може да считаме, че таблицата има два допълнителни реда с номера 0 и $n+1$ и два допълнителни стълба с номера 0 и $k+1$. За клетките, намиращи се в тези редове и стълбове, ще считаме, че са оцветени в бяло. По този начин всяка клетка от оригиналната таблица ще има лява, дясна, горна и долна съседна клетка. При избора на начините за оцветяване на текущата клетка на таблицата има следните възможности:

- клетката може да се оцвети и в черно, и в бяло;
- клетката може да се оцвети само в черно;
- клетката може да се оцвети само в бяло;
- клетката не може да се оцвети нито в черно, нито в бяло.

Решението за възможните начини за оцветяване вземаме, отчитайки информацията, която имаме за горната, лявата и дясната съседни клетки на текущата. Тъй като текущата клетка е последната неоцветена съседна на горната клека, то избора на цвят на текущата клетка гарантира и коректността на условието горната съседна клетка да

има точно една съседна клетка, оцветена в черно. Малко по-особени са следните ситуации:

- за клетките от първия ред на таблицата, горната и дясната съседни клетки на текущата не водят до никакви ограничения при избора на нейното оцветяване;
- за клетките от първия стълб на таблицата лявата съседна клетка на текущата не носи никаква информация за начина на нейното оцветяване;
- за клетките от $n+1$ -я ред на таблицата оцветяване не се извършва, тъй като те се оцветени в бяло. Все пак обаче те трябва да бъдат посетени при генерирането, тъй като при преминаването през тях се извършва проверка дали техните горни съседни клетки (клетките от n-тия ред на таблицата) изпълняват условието за единствена черна съседна клетка.

```
#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;
}
```

Кръг 3 / Задача А4 / В4. ТАБЛИЦА

Най-напред да огледаме понятието за еквивалентност, дефинирано в задачата. Във всеки от класовете на еквивалентност чрез разместване на редове и/или колонки можем да постигнем следното:

- да преместим най-малкото от дванадесетте числа в горния ляв ъгъл на таблицата;
- да подредим числата от първия ред на таблицата нарастващо;
- да подредим числата от първия стълб на таблицата нарастващо.

Ясно е, че ако две таблици са били еквивалентни, при това "канонично" подреждане те ще бъдат едни и същи. И обратно – ако след привеждане в каноничен вид две таблици не са едни и същи, те не са еквивалентни. Значи можем да считаме, че ще създаваме (и ще броим) само каноничните подредби – останалите са еквивалентни на някоя от тях.

Едно от очевидните необходими условия за наличието на правилна подредба е сумата от дванадесетте числа да е кратна на 6 – сумата по редове е четна, а по колонки се дели на 3. Всъщност, остатъците на числата при деление на 6 играят в задачата изключителна роля. Можем, например, да избегнем работата с големи числа, като заменим всяко от тях с по-малко със същия остатък, като все пак запазим отношенията по големина между входните данни (заради каноничното представяне). Така работните данни всъщност могат да не надвишават 71. Най-лесно това се реализира след подреждане на входните данни (например в растящ ред). В разсъжденията по-нататък ще считаме, че входните данни са подредени така (и евентуално заменени с по-малки) в масив `data`, с най-малък индекс 0 и най-голям 11.

След това е важно да направим оценка на най-големия възможен брой канонични таблици.

$data_0$	r_0	r_1	r_2
c_0	$rest_0$	$rest_1$	$rest_2$
c_1	$rest_3$	$rest_4$	$rest_5$

Горният ляв ъгъл е фиксиран. От оставащите 11 стойности трябва да изберем две (c_0 и c_1) за първа колонка, а от оставащите 9 – три (r_0, r_1 и r_2) за първи ред, като образуваме своеобразен „подпис" на всеки клас на еквивалентност – $\{(c_0, c_1), (r_0, r_1, r_2)\}$. Избраните елементи подреждаме по големина, значи редът на избор няма значение. Това прави $\binom{11}{2}\binom{9}{3} = 4620$ възможни избора.

Останалите 6 позиции в таблицата могат да бъдат запълнени от останалите 6 числа по произволен начин и всяка тяхна подредба поражда нов клас на еквивалентност. Това прави за всеки „подпис" по 6!=720 възможности, или общо най-много 4620.720=3326400 възможни класове на еквивалентност. Ако всеки входен елемент може да бъде поставен на всяко място (например, очевидно това е случаят, когато всички числа от входа дават еднакви остатъци при деление на 6), получената горна граница е и крайният резултат.

Добре организиран алгоритъм с връщане (backtracking) може да реши задачата въз основа на горните разсъждения. Ако съумеем и да го оптимизираме, той ще бъде достатъчно ефективен и за най-големите стойности. Ето една динамична идея, която цели да използваме наготово подзадача, която по принцип вече сме решавали.

„Подписите", наредени и взети по модул 6, образуват „редуциран подпис": петцифреното шестично число $\overline{c_0c_1r_0r_1r_2}$. Възможностите са 6^5=7776. Ненаредените остатъци по модул 6 на останалите числа (защото всяко може да бъде на всяко място) определят задачата до еднозначност. Ако разгледаме тези остатъци като брой (b_0 – брой остатъци 0, b_1 – брой остатъци 1, …, b_5 – брой остатъци 5), то ще имаме $b_0+b_1+b_2+b_3+b_4+b_5=6$. Даже и тази доста точна оценка обаче изисква твърде много памет за директно прилагане на динамичната идея (над 13MB, които всъщност няма да използваме ефективно), а и е трудоемка за кодиране. По-разумно е да отделим статичен масив от указатели – по един за всеки „редуциран подпис", който да сочи към динамично оформен списък от решени подзадачи с този „редуциран подпис". Масива от остатъци b ще кодираме „без икономии" (но бързо) в 4 байта.
Може, разбира се, да се приложи и чисто комбинаторна реализация върху същите идеи.

Реализация

```
#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;
}
```

Кръг 3 / Задача А5 / В5. МАТРЬОШКИ

Анализ

На всяка стая в задачата можем да съпоставим координати *(x, y)*, където *x* е редът, в който се намира на входа, а *y* е колоната (броим от *1* до *N* от горе на долу и от ляво надясно). С M_{xy} ще бележим големината на матрьошката в стая *(x, y)*. Нека дефинираме път в лабиринта като последователност от помещения, такива че всяко следващо, освен първото, да съдържа по-голяма матрьошка и координатите му да са не по-малки от тези на предишното. Очевидно Пешо Хакерът може да излезе от лабиринта, като мине през всички стаи от даден път и събере матрьошките в тях. Така ще знаем отговора на задачата, ако намерим дължината на най-дългия път.

Решение

Ще решим задачата по метода на динамичното оптимиране. Нека дефинираме подзадачата *f(i, j)*, която намира най-голямата дължина на път, завършващ в стая *(i, j)*. Стойността на *f(i, j)* можем да намерим рекурентно по формулата $f(i, j)=1+max\{f(k, l) \mid k \le i, l \le j, (k, l) \ne (i, j), M_{kl}<M_{ij}\}$.

Директната реализация на този алгоритъм би била да използваме квадратна матрица, която да съдържа стойностите на *f(i, j)*. За пресмятането на всяка подзадача можем да обходим всички подзадачи, които отговарят на условията от рекурентната формула и да изберем максималната. Това ще ни доведе до сложност по време $O(N^4)$. При дадените ограничения, разбира се, такова решение няма да е достатъчно бързо на всички тестове и ще получи не повече от 50 точки.

Можем да оптимизираме изчисляването на всяка подзадача, като използваме структура с логаритмични времена за обновяване и запитване (например индексно дърво) за да намираме максималните стойности. Нека се върнем на рекурентната формула. Очевидно най-дългият намерен до тук път не може да има отрицателна дължина, следователно търсеният максимум е най-малко нула. В такъв случай, ако първоначално присвоим нули на всички елементи, условието $(k, l) \ne (i, j)$ става излишно. Остават три условия, с които структурата трябва да се съобразява. Тримерното индексно дърво ще ни доведе до доста тежка имплементация и крайна сложност по време $O(N^2.ln^3 N)$ и $O(N^4)$ по памет.

По-добра сложност бихме могли да получим, ако се лишим от още някое условие и съответното му измерение в дървото. Очевидно всяка подзадача се влияе единствено от подзадачите, чиито съответни стаи съдържат по-малки матрьошки. Следователно можем да решаваме подзадачите по реда на нарастване размерите на играчките (те са неповтарящи се). В такъв случай условието $M_{kl}<M_{ij}$ също става излишно, защото в дървото няма да има добавени подзадачи, чиито стаи съдържат по-големи матрьошки от тази на текущата. Двумерното индексно дърво от своя страна ни води и до решение със сложност $O(N^2.ln^2 N)$, което ще получи 100 точки.

Оценяване

Ако решението бъде реализирано достатъчно качествено, то ще получи точки според сложността си: *изчерпване* – 10; $O(N^4)$ – 50; $O(N^2.ln^2 N)$ – 100.

Кръг 3 / Задача C1. ИГРА

Ще използваме два едномерни масива:
масив a, в който $a[i]=0$, когато i-тата клетка е бяла и $a[i]=1$, когато е черна;
масив d, в който $d[i]$ е разстоянието (минималния брох ходове, необходими за придвижването на пионката от началната клетка до текущата клетка); $d[i] = -1$, когато i-тата клетка не е достъпна.

Отначало $d[x]=0$ и $d[i] = -1$ за всички останали клетки. Прилагаме алгоритъма за търсене в ширина. За текущата клетка z сканираме четирите посоки (нагоре, надолу, наляво, надясно) за наличието на непосетени бели клетки. Ако се срещне такава клетка, нейното разстояние трябва да се обнови и номерът на тази клетка да се включи в опашката.

```
#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;
}
```

Кръг 3 / Задача С2. **ТРАНСФОРМАЦИИ**

Нека a има n цифри. Тогава в a има $n-k+1$ групи от последователни цифри, които могат да се променят при един ход. Да номерираме тези групи: първата група да е тази, която съдържа от първата до k-тата цифра на a, втората група – от втората до $(k+1)$-та цифра на a, …, $(n-k+1)$-та група – от $(n-k+1)$-та до n-тата цифра на a. За определяне на минималния брой ходове, с които от a може да се получи b, може да направим следното: като преобразуваме само първата група цифри, извършваме минималния брой ходове, с помощта на които първата цифра на a става равна на първата цифра на b (този брой ходове може да бъде 0, 1, 2, 3, 4, 5, 6, 7 или 8). След това правим същото за новополученото число, като използваме само втората група цифри, така че вече и втората му цифра да съвпада с тази на b. Продължаваме така, докато първите $n-k+1$ цифри на преобразуваното число и на b станат еднакви. Ако след тези действия преобразуваното число и b станат равни, то търсеният минимален брой ходове е този, който сме направили. Ако двете числа са различни, то от a не може да се получи b колкото и хода да направим.
Остава да покажем, че описаният алгоритъм води до верен резултат. Всяка последователност от m на брой хода може да разглеждаме като редица от m числа, всяко от които е между 1 и $(n-k+1)$ и показва с коя група сме извършили поредния ход. Така например, ако a = 12349 и k = 2, редицата 2, 1, 3, 1 означава, че последователно сме получили следните числа: 13449, 24449, 24559 и 35559. Лесно се забелязва, че ако в така описаната редица разменим местата на някои от числата, то крайният резултат няма да се промени (например редицата 1, 1, 2, 3 генерира числата 23349, 34349, 35449 и 35559). Следователно може първо да извършим всички необходими ходове с първата група, след това с втората и т.н.

```
#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;
}
```

Кръг 3 / Задача С3. СТРАННИ ДУМИ

Решението на задачата се състои в организация на цикъл, който чете текста дума по дума до въвеждане на знак за край на текст. За всяка прочетена дума `s` се проверява дали тя е странна и ако е такава, дължината и `d` се сравнява с тази на най-дългата до момента странна дума `md`. Ако намерената дума е с по-голяма дължина – тя заема мястото на най-дългата странна дума `ms=s`.
Проверката за странна дума се изпълнява от функцията sword, която използва стек от знаци. Думата се сканира отляво надясно и за всеки знак от нея се извършва следното:

- Ако стекът не е празен и знакът, записан на върха му съвпада с текущия, от стека се изтрива последния знак.
- В противен случай, текущият знак се записва в стека.

Като вторичен ефект функцията преобразува думата, като заменя всички главни букви с малки.
Да обърнем внимание, че функцията получава като втори параметър дължината на думата, за да се предотврати повторното и пресмятане.
За решението са използвани стандартните типове `string` и `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;
}
```

Кръг 3 / Задача С4. МНОГОЪГЪЛНИК

Лицето на многоъгълника се получава чрез добавяне или изваждане на лицата на правоъгълниците, образувани от хоризонталните страни и оста *x*. Лицето на един правоъгълник се добавя към лицето на многоъгълника, ако вертикална права, преминаваща през точка *(x+0.5, y+0.5),* където *x* и *y* са координати на левия край на съответната страна, пресича нечетен брой от останалите страни на многоъгълника. Ако броят на пресичанията е четен, лицето на правоъгълника се изважда.

```
#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;
}
```

Кръг 3 / Задача С5. ВЕЗНА

Числото *m* се представя като сума $d[0]*3^0 + d[1]*3^1 + d[2]* 3^2 + d[3]* 3^3 + \ldots + d[k]3^k$, където коефициентите d[i] приемат стойности –1, 0 или 1. На лявото блюдо на везната се поставят предметът с маса *m* и теглилките, пред чиито маси стоят коефициенти –1, а на дясното блюдо – теглилките, пред чиито маси стоят коефициенти 1. Теглилките, пред чиито маси стои коефициент 0, не се използват.

```
#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;
}
```

Кръг 3 / Задача С6. ТАНЦИ

Очевидно, ако x или y са нечетни числа момичето или няма да бъде достигнато или ще бъде задминато от момчето.

Тъй като една танцова стъпка се състои от две крачки променяме x и y, съответно с x/2 и y/2.

Някои участници биха разпознали модела на задачата, в която се търсят всички пътища (0,0) до (x,y) с позволени движения само нагоре или надясно, което е биномния коефициент C(x+y,x). Изчисляването му обаче с използване на рекурсия би било грешно заради зададените ограничения за време.

И така общият брой на пътищата, стигащи до момичето, започващи от позиция на момчето (i, j) се намира като сума от пътищата започващи от позиция (i-1, j) и тези, започващи в (i, j-1).

Използвайки техниката на динамично програмиране това число може да бъде пресметнато с използването на двумерен масив или още по-ефективно с използването на едномерен масив, където всяко число се получава от числото, записано преди това на това място j и числото съхранено на място j-1 в масива.

```
#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;
  }
```

Кръг 3 / Задача D2. СЪКРОВИЩЕ

Записаните на листа *N* реда със знаци можем да разглеждаме като елементи на масив от низове. За всеки елемент на масива намираме най-малкия знак. В програмата функцията **char** search(string s, **int** m) връща намерения най-малък знак, който не се среща в даден низ s и е по-голям от най-малкия знак m в този низ, или точка (**'.'**), ако низът s съдържа всички латински букви. За определяне на принадлежността на даден знак към низа се използва помощен масив b със 123 елемента, тъй като по условие използваните знаци са малки и главни латински букви, а най-големият от тези знаци е 'z' с ASCII код 122. Масивът b се нулира в началото на функцията. Всеки негов елемент ще има стойност 0, ако знакът със съответния код не е в низа и 1 – в противен случай. Изключваме знаците, които се намират между главните и малки латински букви като на масив b присвояваме стойности 1 за кодовете на тези знаци. След като се запълни масивът b, вече можем да определим кой е търсения знак като намерим първия елемент със стойност 0. Проверката правим като започнем от намерения най-малък знак m в низа.

```
#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;
}
```

Кръг 3 / Задача D3. ЦИФРА

Моделираме процеса с използването на средствата за низови потоци в C++. В първия цикъл поставяме всички цели числа от 1 до 3000 в буфер *a*, който е низов поток. След това извличаме от буфера *n* цифри една по една.

```
#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;
}
```

Кръг 3 / Задача E1. РОМБОВЕ

Програмата трябва да въведе едно цяло число N, което показва колко вписани ромбчета има фигурата. Броят на редовете, които трябва да се изведат е 2N+1. Всеки ред се състои от две части: определен брой празни позиции (sp) и определен брой (br) *. Лесно се вижда, че на първия ред имаме N празни позиции и една звездичка.

1. Посредством първия цикъл for изчертаваме първите N реда от фигурата, в които броят на празните позиции намалява с единица за всеки следващ ред, а броят на звездичките се увеличава с 2.
2. Следва ред, който се състои от N звездички, едно празно и N звездички:

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

3. Посредством следващия цикъл for изчертаваме следващите N реда, при които броят на празните позиции се увеличава с единица за всеки следващ ред, а броят на знаците намалява с 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;
    }
}
```

Кръг 3 / Задача Е3. НЕСЪКРАТИМА ДРОБ

От зададените граници $0 < a < b < 1000$ следва, че всеки общ делител на числителя и знаменателя е по-малък или равен на 500. Затова опитваме всички възможности за d от 2 до 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;
}
```