НАЦИОНАЛНА ОЛИМПИАДА ПО ИНФОРМАТИКА
Общински кръг, 27 януари 2007 г.

Автори на задачите

A1 – Красимир Манев A2 – Красимир Манев A3 – Красимир Манев	D1 – Венета Богданова D2 – Венета Богданова D3 – Галина Момчева
B1 – Младен Манев B2 – Николина Николова B2 – Емил Келеведжиев	E1 – Бисерка Йовчева E2 – Велислава Христова E3 – Велислава Христова
C1 – Павлин Пеев C2 – Павлин Пеев C3 – Каталина Григорова	

Задача A1. РАЗЛИКА

На стандартния вход е зададено цяло положително число R, не по-голямо от 4100000000. Напишете програма **`DIF`**, която намира и извежда на стандартния изход две цели неотрицателни числа N и M, такива че сумата на целите числа от 1 до N, минус сумата на целите числа от 1 до M да е равна на R. Ако съществуват няколко двойки N и M с исканото свойство, програмата трябва да изведе онази от тях, за която N е най-малко. Входните данни са такива, че сумата на целите числа от 1 до N не надхвърля 4100000000.

ПРИМЕР 1

Вход	**Изход**
`6`	`3 0`

ПРИМЕР 2

Вход	**Изход**
`61`	`31 29`

Решение:

Да означим с S_K сумата на целите числа от 1 до K. Погледнато математически, задачата винаги има решение защото $S_R - S_{R-1} = R$. Тривиалната идея да се използва формула за сумите S_N (на числата от 1 до N) и S_M (на числата от 1 до M) и да се сравнява $S_N - S_M$ с R, очевидно е приложима само за неголеми стойности на N. А за тези, за които е приложима, тази идея ще ни даде алгоритъм със сложност $O(N^2)$.

За да получим алгортъм със сложност $O(N)$ прилагаме достатъчно общата схема, която ще наричаме „плъзгащ се прозорец”. В началото образуваме прозореца, като пресмятаме сумата на първите K числа, където K е най-малкото цяло за което $S_K \geq R$. Ако $S_K = R$, тогава $N = K$, $M = 0$ е търсеното решение. В противен случай започваме „плъзгане” на получения прозорец $[M,N]$. Докато $S_N - S_M > R$, увеличаваме M с 1. Когато $S_N - S_M$ стане по-малко от R, докато $S_N - S_M < R$, увеличаваме N с 1. Когато за пръв път $S_N - S_M$ стане равно на R, текущите стойности на N и M са търсеното решение. При зададените параметри, стойностите на S_N и S_M трябва да се съхраняват в подходящ тип (най-малко `unsigned int` за пишещите на C/C++).

```
#include <stdio.h>
main()
{
   unsigned N=0,M=0,SN=0,SM=0,R;
   scanf("%d",&R);
   while(SN+N<R){SN+=N;N++;}SN+=N;
   if(SN>R)
      while(SM+M<SN-R){SM+=M;M++;}
   SM+=M;
   if(SN-SM==R) printf("%d %d\n",N,M);
   else
   {
      while(N<=R)
      {
         N++;
         //if(SN+N<SN){printf("Not calculable!\n");break;}
         SN+=N;M++;
         while(SM+M<SN-R){SM+=M;M++;}
         SM+=M;
         if(SN-SM==R){printf("%d %d\n",N,M);break;}
      }
   }
}
```

Решението има един недостатък. То няма да даде искания резултат, ако някоя от сумите S_N и S_M надхвърли максималната стойност на използвания тип. Затова е ограничението, всяка една от сумите да не надхвърля 4100000000. Тази стойност, обаче, е условна, за да се избегне явното указване на необходимия тип. Забележете, че в решението (поставения като коментар ред) се използва максимално възможностите на типа `unsigned int`. Така в множеството от тестове се е „промъкнал" тест при който $S_N > 4100000000$, което не е проблем. Ако някои състезатели са следвали буквално условието и затова не са получили решение на този тест, редно е да се пусне допълнителен тест, като за да се намери такъв, трябва да се промени условието `SN+N<SN` на `SN+N>4100000000`.

Задача А2. НИЗОВЕ

Низът $\alpha = a_1 a_2 ... a_P$ е *начало* на низа $\beta = b_1 b_2 ... b_Q$, ако $0 < P < Q$ и $a_K = b_K$, $K = 1,2,\ldots,P$. На първия ред на стандартния вход е зададено цялото положително число N, $N \le 1000$. На всеки от следващите N реда е зададен по един низ, съставен от малки латински букви, с дължина L, $1 \le L \le 20$. Напишете програма **ORD**, която намира и извежда на стандартния изход дължината M на най-дългата редица $\alpha_1, \alpha_2, \ldots, \alpha_M$, съставена от някои от зададените низове такава, че α_K е начало на α_{K+1}, $K = 1,2,\ldots,M-1$.

ПРИМЕР

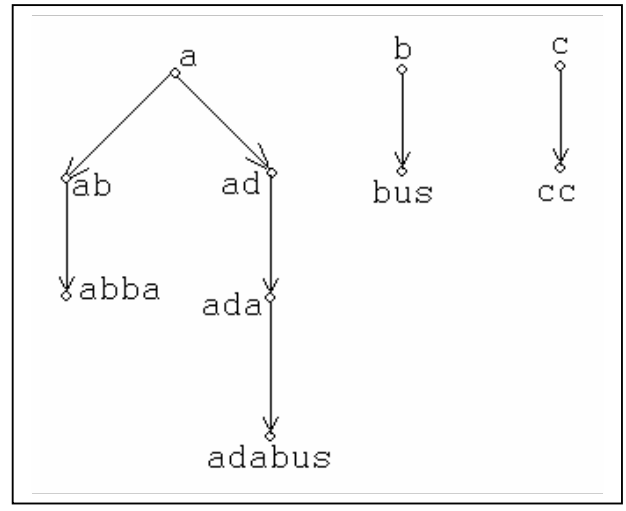
Вход	**Изход**
10	4
a	
b	
ad	
ab	
ada	
c	
adabus	
bus	
abba	
cc	

Решение:

Очевидно е, че релацията между низове „*е начало на*", дефинирана в задачата, е преднаредба или, което е едно и също, ориентиран ацикличен граф (*даг*) $G(V,E)$. Поради по-голямата популярност на второто понятие, ще си служим с него в това обяснение. Задачата, поставена в условието, сега може да се формулира така: по зададен даг да се намери броят на върховете на един негов най-дълъг път. Популярен подход за решаване на тази задача е да се използва динамично програмиране. За всеки връх v на дага означаваме с P_v дължината на максималния път, завършващ във v. За всеки връх v, който няма предшественици в дага, $P_v = 1$. Ако преките предшественици на v в дага са $u_1, u_2, \ldots, u_K$, тогава

$$P_v = \max \{ P_{u1}, P_{u2}, \ldots, P_{uK} \} + 1$$

За да се извърши пресмятането по-лесно, добре е предварително да се сортират топологически върховете на дага и пресмятанията да се извършват в реда на топологическото сортиране. Така, при пресмятане на стойността P_v за един връх v, стойностите на всички негови предшественици ще са вече пресметнати. Неприятното в това решение е необходимостта да се построи дага, което ще отнеме $O(N^2)$ стъпки. Топологическо сортиране, реализирано с обхождане на дага в дълбочина, ще добави още $O(M)$ стъпки, където M е броят на ребрата на дага. Самото изчисляване на стойностите и намиране на максималната също ще изисква $O(M)$ стъпки. Така сложността на алгоритъма ще бъде $O(N^2) + 2.O(M) = O(N^2)$, защото броят на ребрата M е $O(N^2)$.



Да опитаме нещо друго. В сила е следното

Твърдение. *Лексикографското сортиране на множество от низове е топологическо сортиране на върховете на съответния даг.*

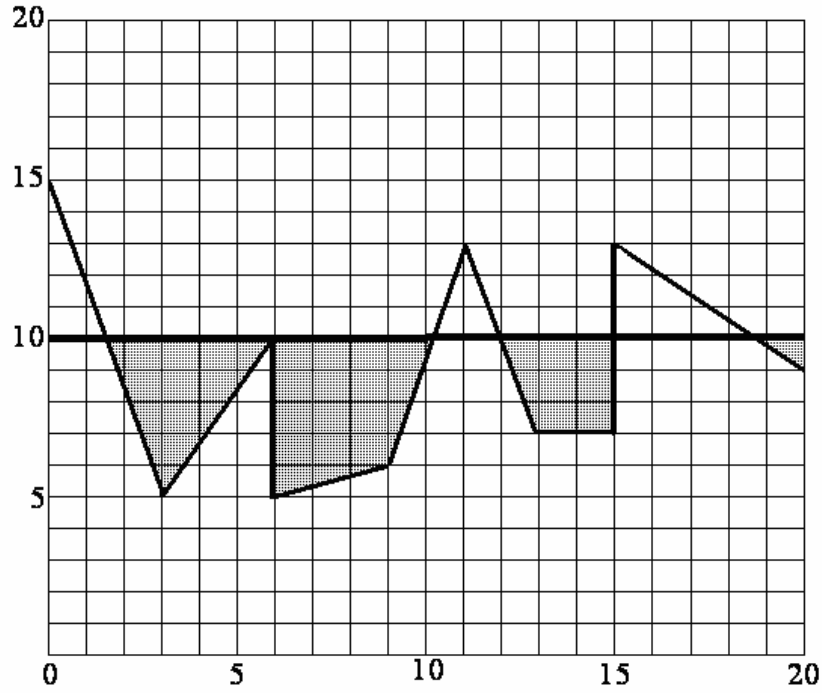
доказателството на което оставяме на читателя за упражнение. Нещо повече, ако представим в даг само преките предшествания, ще получим гора от ориентирани дървета. За примера от условието, съответната гора е показана на фигурата.

Да сортираме лексикографски низовете (в приложената програма не е фиксирано как ще сортираме). Заблежете, че отделните дървета на гората заемат плътни участъци от лексикографското сортиране. Остава да направим обхождането в дълбочина на сортираните топологически низове (вж. Твърдението), да пресметнем необходимите ни стойности P_v, като попътно намерим и максималната от тези стойности. Но това вече става със сложност $O(N)$, защото всеки връх минава точно един път през стека (определящ за това е фактът, че графът на непосредствените прешественици е гора от дървета, а не произволен даг). Така сложността на алгоритъма е равна на сложността на лексикографското сортиране. При използване на сортиране със сложност $O(N.\log N)$ ще получим много по-бърз алгоритъм. При алгоритъм за сортиране със сложност $O(N^2)$ порядъците на двата алгоритъма са еднакви, и все пак вторият ще бъде по-бърз. За нуждите на състезанието, „бързият" библиотечен `quick_sort` е напълно достататъчен.

```
#include <stdio.h>
#include <string.h>
char a[1001][21],t[21];
int s[1001],sp;
int prefix(int x,int y)
{
    int i=0;
    while(a[x][i]==a[y][i])i++;
    if(a[x][i]=='\0') return 1;
    else return 0;
}
main()
{
   int N,i,j,max;
   scanf("%d",&N);
   for(i=1;i<=N;i++) scanf("%s",a[i]);
   < тук е мястото на сортирането >
   s[1]=1;sp=1;max=1;
   for(i=2;i<=N;i++)
   {
      if(prefix(i-1,i))
      {s[i]=s[i-1]+1;if(max<s[i])max=s[i];}
      else
      {  j=i-2;
         while(j>=sp&&!prefix(j,i))j--;
         if(j<sp){s[i]=1;sp=i;}
         else {s[i]=s[j]+1;if(max<s[i])max=s[i];}
      }
   }
   printf("%d\n",max);
}
```

Задача А3. ПЯСЪЧЕН БРЯГ

Във връзка с влизането на България в ЕС, арендаторът Аре Балкански решил да направи голяма инвестиция и да „пооправи" бреговата линия на арендувания от него плаж. Линията на брега, поставена в координатна система, може за се опише като полигон, съставен от N отсечки, $1 \le N \le 1000$, който не се самопресича.



За подравняването той избрал правата $y = C$, успоредна на абсцисната ос. Преди да пусне багерите, останало само да определи количеството пясък, което ще се отстрани. За целта е нужна програма **SAND**, която намира и извежда на стандартния изход лицето S на засегнатия участък (защрихован на фигурата), закръглено до четвъртия знак след десетичната точка. На първия ред на стандартния вход са зададени целите числа N и C, а на всеки от следващите $N + 1$ – поредната двойка координати x_i и y_i на връх на полигона, като $x_i \le x_{i+1}$. Координатите са цели числа, в интервала от -10^6 до 10^6, а стойността на търсеното лице се побира в рамките на типа `double`.

ПРИМЕР

Вход	**Изход**
9 10 0 15 3 5 6 10 6 5 9 6 11 13 13 7 15 7 15 13 20 9	35.1607

Решение:

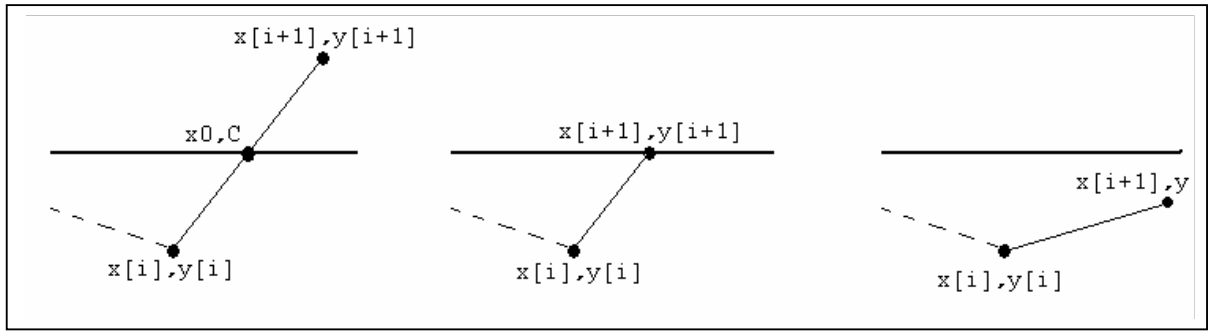
Както в повечето задачи на изчислителната геометрия, и в тази задача може да се забележи наличието на много различни случаи, които трябва да бъдат разгледани. Пропускането на някой от случаите би могъл да доведе до неточно решение. Затова трябва да бъдат пълно и точно класифицирани всички случаи. Един начин да направим това е с помощта на краен автомат. Без да влизаме в подробности (желаещите да се запознаят по-подробно с понятието могат да го направят в кой да е учебник по Дискретна математика) ще кажем, че крайните автомати, които ще използваме, се определят с множеството Q от *състоянията* си, множеството V от *входовете* си, множеството W от *изходите* си и правило, по което от текущото състояние q и вход v се определя следващото състояние q' и изхода w.

Ще разгледаме автомат с две състояния: 0 – поредният връх на полигона се намира над или върху разделителната линия $y = C$ и тази част от полигона не участва в пресмятаното лице; 1 – поредният връх на полигона се намира под разделителната линия и тази част от полигона участва в пресмятаното лице. В началото автоматът е в състояние 0. Входовете на автомата ще бъдат 9 (кодирани с числата от 0 до 8) и се определят от положението на краищата на текущата отсечка спрямо разделителната линия. Ако левият край е над разделителната линия, то входът ще бъде 0, 1 или 2 в зависимост от това дали десният край а над, върху или под разделителната линия. Аналогично, случаите с ляв край на текущата отсечка върху разделителната права кодираме с 3, 4 или 5, а случаите с ляв край на текущата отсечка под разделителната права – с 6, 7 или 8. Функцията `calc_input`, по зададен в променливата `i` номер на поредната отсечка на полигона, пресмята в променливата `j` съответната стойност на входа, а в променливата `x0` пресмята, ако е необходимо, x-координатата на пресечната точка на отсечката с разделителната линия.

В таблицата е дефиниран съответният автомат, като за всяко от състоянията и всеки от входовете са покзани следващото състояние и (след знака „;”) съответният изход. Отсъствието на изход е показано с „–”.

	вх. 0	вх. 1	вх. 2	вх. 3	вх. 4	вх. 5	вх. 6	вх. 7	Вх. 8
съст. 0	0; –	0; –	1; w_1	0; –	0; –	1; w_2	0; w_3	0; w_4	1; w_5
съст. 1	1; –	1; –	1; –	1; –	1; –	1; –	0; w_6	0; w_7	1; w_8

Изходите на автомата въщност са кодът който трябва да се изпълни, при съответно стояние и вход. Те се състоят от добавяне в масива `p` (с индекс `L`) на координатите на една или повече точки и евентуално пресмятане (с функцията `calc_face`) на лицето на поредното „парче”, ако такова парче вее се е формирало. Ще разгледаме по-подробно дефиницията на автомата в състояние 1 и ще оставим на читателя за упражнение дефиницията в състояние 0. Ако се намираме в състояние 1, т.е. под разделителната линия, тогава входовете 0, 1 и 2 (левият край на отсечката е над разделителната линия) и входовете 3, 4 и 5 (левият край на отсечката е върху разделителната линия) са невъзможни. Затова можем да считаме, че автомат остава в същото състяние и не е нужно дефинирането на изход. За случаите 6, 7 и 8 виж Фигурата



Във всички случаи точката с координати (`x[i],y[i]`) вече е в масива `p`, затова в случай 6 (w_6) поставяме в него и точката с координати (`x0,C`) . Това завършва поредното парче. Добавяме в края и началната точката на парчето с координати (`p[X][0],p[Y][0]`) и пресмятаме лицето му. Тъй като краят на отсечката е над разделителнта линия, сменяме

състоянието на 0. Случаят 7 (w_7) е аналогичен, но вместо (x0,C) поставяме в масива p координатите (x[i+1],y[i+1]). В случая 8 (w_8) поставяме в масива p само координатите (x[i+1],y[i+1]). Те не завършват парче, затова не пресмятаме лице, а тъй като краят на отсечката остава под линията, не сменяме и състоянието.

```
#include <stdio.h>
#define X 0
#define Y 1
#define MAXN 2005
int N; double C; double x[MAXN],y[MAXN],p[2][MAXN];
void calc_input(int i,int* j,double* x0)
{    int a,b; double d;
     if(y[i]>C) a=0;
     else {if (y[i]==C) a=1; else a=2;}
     if(y[i+1]>C) b=0;
     else {if (y[i+1]==C) b=1; else b=2;}
     *j=3*a+b;
     if(*j==2||*j==6)
     {
        if(x[i]==x[i+1]) *x0=x[i];
        else
        {
           d=(x[i+1]-x[i])/(y[i+1]-y[i]);
           *x0=x[i]+(C-y[i])*d;
        }
     }
}
double calc_face(int L)
{  double s=0.;int i;
   for(i=1;i<=L;i++)
      s+=(p[Y][i]+p[Y][i-1])*(p[X][i]-p[X][i-1]);
   if(s<0) s= -s;
   return s/2.;
}
main()
{
    int i,j,L=0,state=0; double face=0.,x0,y0;
    scanf("%d %lf",&N,&C);
    for(i=0;i<=N;i++) scanf("%lf %lf",&x[i],&y[i]);
    for(i=0;i<N;i++)
    {  calc_input(i,&j,&x0);
       if(state==0)
           switch(j)
           {   case 2: p[X][L]=x0;p[Y][L++]=C;
                       p[X][L]=x[i+1];p[Y][L++]=y[i+1];
                       state=1;
                       break;
               case 5: p[X][L]=x[i];p[Y][L++]=y[i];
                       p[X][L]=x[i+1];p[Y][L++]=y[i+1];
                       state=1;
                       break;
```

```
                case 6: p[X][L]=x[i];p[Y][L++]=C;
                        p[X][L]=x[i];p[Y][L++]=y[i];
                        p[X][L]=x0;p[Y][L++]=C;
                        p[X][L]=x[i];p[Y][L]=C;
                        face+=calc_face(L);L=0;
                        break;
                case 7: p[X][L]=x[i];p[Y][L++]=C;
                        p[X][L]=x[i];p[Y][L++]=y[i];
                        p[X][L]=x[i+1];p[Y][L++]=y[i+1];
                        p[X][L]=x[i];p[Y][L]=C;
                        face+=calc_face(L);L=0;
                        break;
                case 8: p[X][L]=x[i];p[Y][L++]=C;
                        p[X][L]=x[i];p[Y][L++]=y[i];
                        p[X][L]=x[i+1];p[Y][L++]=y[i+1];
                        state=1;
                        break;
            }
        else
            switch(j)
            {
                case 6: p[X][L]=x0;p[Y][L++]=C;
                        p[X][L]=p[X][0];p[Y][L]=p[Y][0];
                        face+=calc_face(L);L=0;
                        state=0;
                        break;
                case 7: p[X][L]=x[i+1];p[Y][L++]=y[i+1];
                        p[X][L]=p[X][0];p[Y][L]=p[Y][0];
                        face+=calc_face(L);L=0;
                        state=0;
                        break;
                case 8: p[X][L]=x[i+1];p[Y][L++]=y[i+1];
                        break;
            }
    }
    if(state==1)
    {
       p[X][L]=x[N];p[Y][L++]=C;
       p[X][L]=p[X][0];p[Y][L]=p[Y][0];
       face+=calc_face(L);
    }
    printf("%lf\n",face);
    return 0;
}
```

Задача B1. СТЕПЕН

Напишете програма **POWER**, която въвежда от един ред на стандартния вход три цели числа n, k и p ($10^4 < n < 10^9$, $0 < k < 10^9$, $0 < p < 5$) и извежда на стандартния изход последните p цифри на n^k.

ПРИМЕР
Вход
```
1001 4 3
```
Изход
```
001
```

Решение:

Прилагането на наивния метод на намиране на степен чрез последователно умножение ще даде решение при не повече от 70% от тестовете. За да се справим с всички тестове трябва да се програмира метод за бързо степенуване. Например, за да намерим стойността n^k, може да използваме рекурсия, като вземем предвид, че:

- ако $k=0$, то $n^k=1$;
- ако k е четно естествено число, то $n^k=n^{\frac{k}{2}}.n^{\frac{k}{2}}$;
- ако k е нечетно естествено число, то $n^k=n^{\frac{k-1}{2}}.n^{\frac{k-1}{2}}.n$.

Тъй като се интересуваме само от последните p цифри на n^k, то е достатъчно да се работи само с последните p цифри на получаваните междинни резултати.

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

int n,k,p,d,n1;

int power(int k)
{
  if (k==0) return 1;
  if (k%2==0) {n1=power(k/2); return n1*n1%d;}
  else {n1=power(k/2); return n1*n1%d*n%d;}
}

int main()

{
  cin >> n >> k >> p;
  switch(p)
  {
    case 1: d=10;    break;
    case 2: d=100;   break;
    case 3: d=1000;  break;
    case 4: d=10000; break;
  }
  n=n%d;
  n=power(k);
  for (int i=1; i<p+1; i++)
    { d=d/10;
      cout << n/d;
      n=n-n/d*d;
    }
  cout << endl;

  return 0;
}
```

Задача B2. ЗАПЛАЩАНЕ

Както е известно, поради проблеми със софтуерната система, обслужваща клиентите на електроразпределение, около Нова година опашките за заплащане на потреблението на електричество във Велико Търново нараснаха неимоверно. Хората се притесняваха, че поради неплатени сметки могат да останат без електрозахранване точно за празниците. Накрая въпросът беше решен, но пък възникна друг проблем – касите не можеха да се справят с бързащите да платят сметките си клиенти. Опашките нарастваха постоянно, започнаха скандали, на места се стигна даже до бой. За разрешаване на проблема с чакащите да платят клиенти, на всеки клиент още при пристигането му се дава входящ номер. За да се спазва предимството на майките с малки деца и възрастните хора, входящият номер се състои от главна латинска буква и поредния номер на клиента. Буквата може да бъде: *M* – майка с малко дете или бременна жена; *O* – възрастен човек; *R* – обикновен гражданин.

Клиентите се обработват по реда на пристигането си, като се дава предимство на възрастните хора пред обикновените граждани и на майките с деца – пред обикновените граждани и пред възрастните хора. Известно е, че докато касата обработи един клиент, на опашката се натрупват още 3 нови клиента. При отваряне на гишето за плащане има вече опашка от 10 чакащи клиента. На ден се обработват по не повече от 1 000 клиента.

Напишете програма **PAYMENT**, която намира поредния номер, под който е обслужен даден клиент.

От първия ред на стандартния вход се въвежда номера на пристигане на даден клиент. От следващия ред се въвеждат входящите номера на клиентите в реда на постъпването им. Номерата са разделени с по един интервал. На стандартния изход се извежда номера, под който е обслужен клиента.

ПРИМЕР

Вход

```
12
R1 R2 O3 R4 R5 R6 O7 R8 R9 O10 M11 M12 R13 M14 O15
```

Изход

```
3
```

Решение:

Задачата е типичен пример за използване на структура Опашка. В случая трябва да се генерират три опашки – по една за всеки тип клиент.

Данните постъпват последователно от входния файл и всеки номер се поставя в съответната опашка в зависимост от буквата пред него. В началото се прочитат последователно 10 поредни номера, а след това се изважда един номер от някоя от опашките и се четат по 3 нови номера.

Изваждането на номер от опашка става като последователно се сканират трите опашки според зададения в задачата приоритет. В дадена опашка се търси номер, само ако опашките с по-висок приоритет са празни. Процесът приключва при намиране на търсения елемент.

```
#include<iostream>
#include<cstdlib>
#include<queue>

using namespace std;

void Place(char num[]);
//Postavia poreden nomer v syotvetnata opashka

queue<int> M, O, R;
```

```
int main(){
     bool b;

     char num[7];
     int client, current, cnt=0;

     cin>>client;
     for (int i=0; i<10; i++){
          cin>>num;
          Place(num);
     }

     do {
          b = 0;
       //Kontrolira cikyla.
        //Ima stojnost 1, ako tyrseniat element ne e nameren
       //i opashkite ne sa prazi
        //i stojnost 1 - v protiven sluchaj -
        //nameren e element ili opashkite sa prazni
          if (!M.empty()) {current = M.front(); M.pop(); cnt++;
b=1;}
          else
               if (!O.empty()){current = O.front(); O.pop();
cnt++; b=1;}
               else
                    if (!R.empty()){
                    current = R.front(); R.pop(); cnt++; b=1;}

          if (b)
           if(current==client){
              cout<<cnt<<endl;
              b=0;
           }
           else
                 for (int i=0; i<3; i++){
                      cin>>num;
                      if (!cin.eof())
                      Place(num);
            }
     }while(b);

     return 0;
}

void Place(char num[]){
     char letter;
     letter = num[0];
     switch (letter){
          case 'M': M.push(atoi(num+1));
                         break;
          case 'O': O.push(atoi(num+1));
                         break;
          case 'R': R.push(atoi(num+1));
                         break;
     }
}
```

Задача В3. ТРИЪГЪЛНИЦИ

В равнината са дадени N триъгълника ($0 < N < 10\,000$) с целочислени координати на върховете си. Напишете програма **TR**, която извежда на стандартния изход броя на нееднаквите триъгълници. Входните данни се четат от стандартния вход, като на първия ред е дадена стойността на N. Следват N реда, всеки съдържащ по три двойки цели числа, разделени с интервал. Всяка двойка задава координатите на абсцисата и на ординатата на връх от поредния триъгълник. Координатите не надминават по абсолютна стойност 1000.

ПРИМЕР

Вход

```
3
0 0 1 0 0 2
0 0 2 0 0 1
0 0 1 0 0 1
```

Изход

```
2
```

Решение:

Два триъгълника са еднакви тогава и само тогава, когато имат съответно равни страни. В програмата последователно се прочитат координатите на върховете на поредния триъгълник от входния поток и се пресмятат квадратите от дължините на страните му. Тези стойности се нареждат в ненамаляващ ред и се извършва проверка, дали тази тройка числа е вече записана в използваната структура данни в програмата. Ако не е записана, тройката се добавя. Накрая се извежда броят на добавените тройки.

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

const int N=10001;
int n;
struct tr{int a, b, c;};
tr d[N];

int main()
{ cin >> n;
  int k=0;
  for(int i=1; i<=n; i++)
  {
    int ax,ay,bx,by,cx,cy;
    cin >> ax >> ay >> bx >> by >> cx >> cy;
    tr p;
    p.a=(cx-bx)*(cx-bx)+(cy-by)*(cy-by);
    p.b=(cx-ax)*(cx-ax)+(cy-ay)*(cy-ay);
    p.c=(ax-bx)*(ax-bx)+(ay-by)*(ay-by);
    if(p.a<p.b)swap(p.a,p.b);
    if(p.a<p.c)swap(p.a,p.c);
    if(p.b<p.c)swap(p.b,p.c);
    bool f=true;
    for(int j=1;j<=k;j++)
     if((d[j].a==p.a)&&(d[j].b==p.b)&&(d[j].c==p.c))
      {f=false;break;}
    if(f){k++;d[k]=p;}
  }
  cout << k << endl;
}
```

Задача С1. ПОДДУМИ

Разглеждаме думи, образувани от главни латински букви.

Последователност от една или повече съседни букви в думата наричаме „поддума”.

Например, някои от думите, които са поддуми на думата ТАРАТОР са: Т, А, АРАТ, ТОР, ТАРАТОР.

Напишете програма **SUBWORDS**, която намира броя на ***различните*** поддуми, които съдържа дадена дума.

От стандартния вход се въвежда един ред, който съдържа една дума с дължина, не по-голяма от 100. На стандартния изход да се изведе един ред с едно естествено число – броя на различните поддуми във въведената дума.

ПРИМЕР

Вход	**Изход**
CTPACT	18

Обяснение: В думата СТРАСТ има 18 различни поддуми: С, Т, Р, А, СТ, ТР, РА, АС, СТР, ТРА, РАС, АСТ, СТРА, ТРАС, РАСТ, СТРАС, ТРАСТ и СТРАСТ.

Решение:

Да се намерят всички поддуми, като се запазят в масив (или по-добра структура) неповтарящите се от тях е едно стандартно решение, чиято коректна реализация решава проблема при тези ограничения. Тук предлагаме едно съвсем икономично на памет решение. При него от максималния възможен брой поддуми се изважда броя на тези, които се повтарят.

Решение (C):

```
#include <stdio.h>
#include <string.h>
char w[128];
// Функция, която определя дали масив от символи m с дължина l
// се среща в С-низа s.
int memstr(const char *m,int l,const char *s)
{char b[128];
 memcpy(b,m,l);
 b[l]=0;
 return strstr(s,b)?1:0;
}
//Функция, която преброява неповтарящите се поддуми в С-низа s.
int Count(const char *s)
{int c,i,j,l=strlen(s);
 c=l*(l+1)/2; //Максимален брой поддуми: една с дължина l,
// 2 с дължина l-1,…, l с дължина 1, т. е. 1+2+3+…+l=l*(l+1)/2.
 for (i=1;i<l;i++)
  for (j=0;j<=l-i;j++)
   if (memstr(&s[j],i,&s[j+1])) c--;//Ако се среща и по-надясно,
                                    // намаляваме броя.
 return c;
}
int main (void)
{fgets(w,102,stdin);
 w[strlen(w)-1]=0;
 printf("%d\n",Count(w));
 return 0;
}
```

Решение (Pascal):

```
VAR
 w:String;
Function memstr(const s,w:string):Boolean;
Begin
 memstr:=pos(s,w)>0;
End;
Function Count(const s:String):Integer;
Var c,i,j,l:Integer;
Begin
 l:=length(s);
 c:=l*(l+1) div 2;
 For i:=1 To l-1 Do
  For j:=1 To l-i Do
   If memstr(copy(s,j,i),copy(s,j+1,255)) Then Dec(c);
 Count:=c;
End;
BEGIN
 ReadLn(w);
 WriteLn(Count(w));
END.
```

Задача С2. ДЪЛБОЧИНА

Разглеждаме следния алгоритъм за преобразуване на естествено число ***n***.

Стъпка 1. Ако числото е 1 – край на алгоритъма.
Стъпка 2. Ако числото е четно – делим го на 2 и се връщаме към Стъпка 1.
Стъпка 3. Умножаваме числото по 3, прибавяме 1 и преминаваме към Стъпка 1.

Да приложим алгоритъма за ***n***=7.

Стъпка 1: 7 не е равно на 1, преминаваме към Стъпка 2
Стъпка 2: 7 не е четно, преминаваме към Стъпка 3
Стъпка 3: 3*7+1=22, преминаваме към Стъпка 1
Стъпка 1: 22 не е равно на 1, преминаваме към Стъпка 2
Стъпка 2: 22 е четно, 22/2=11, преминаваме към Стъпка 1
Стъпка 1: 11 не е равно на едно ... и т. н.

Така числото 7 преминава през следните „състояния”, докато се преобразува в 1:

7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1.

Под „**дълбочина**” на естественото число ***n*** ще разбираме броя на числата в тази редица.

Така дълбочината на 7 е 17. Лесно се пресмята, че дълбочината на 1 е 1, на 2 е 2, на 3 е 8 и т. н.

Напишете програма **DEPTH**, която намира какъв е максималният брой числа с равна дълбочина в зададен интервал от естествени числа, като въвежда от стандартния вход две естествени числа ***a*** и ***b*** ($1 \le a \le b \le 10000$) и извежда на стандартния изход един ред с едно естествено число – максималния брой естествени числа в интервала [***a***, ***b***] с една и съща дълбочина.

ПРИМЕР

Вход	**Изход**
28 70	4

Обяснение: В интервала [28, 70] най-голяма група образуват числата с дълбочина 20, които са четири на брой: 56, 58, 60 и 61.

Решение:

Поставените ограничения не налагат използване на по-дълбоки динамични идеи, макар че би могло. В решението по-долу се използва само масив от броячи L за намерените дълбочини, от които накрая се търси стандартен максимум.

Решение (C++):

```
#include <iostream.h>
unsigned char L[10000]={0};
int a,b,max=0;
long step(long a)
{if (a&1) return 3*a+1;
 return a>>1;
}
long depth(long a)
{long c=1;
 while (a!=1)
 {a=step(a);
  c++;
 }
 return c;
}
void tab(long st, long en)
{int m;
 for (long i=st;i<=en;i++)
 {m=depth(i);
  L[m]++;
  if (m>max) max=m;
 }
}
int main (void)
{int m;
 cin>>a>>b;
 tab(a,b);
 m=0;
 for (int i=1;i<=max;i++) if (L[i]>m) m=L[i];
 cout<<m<<endl;
 return 0;
}
```

Задача С3. ДВОИЧНА СУМА

Двоичната сума на цели положителни числа се получава чрез сумиране цифрите от двоичното им представяне по модул 2. Полученият резултат е двоичното представяне на двоичната сума.

Например, двоичната сума на числата 8, 15, 12 и 2 е 9:

	2^3	2^2	2^1	2^0	
8 =	1	0	0	0	
15 =	1	1	1	1	
12 =	1	1	0	0	
2 =	0	0	1	0	
	1	0	0	1	= 9

Напишете програма **BSUM**, която намира двоичната сума на зададени числа, като въвежда от първия ред на стандартния вход едно цяло число ***n*** ($1<n<=10^6$), а от следващите ***n*** реда – числата, чиято двоична сума трябва да се намери. Числата не надхвърлят 10^9. На стандартния изход се извежда едно цяло число – намерената сума.

ПРИМЕР

Вход	**Изход**
4 8 15 12 2	9

Решение:

Зададените ограничения подсказват, че е необходимо да въвеждаме числата едно по едно и последователно да ги добавяме към сумата. Търсената сума получаваме в масив S, елементите на който съдържат цифрите на нейното двоично представяне. За целта занулляваме елементите на масива S, въвеждаме числата едно по едно, преобразуваме всяко в двоичен вид и го добавяме към сумата.

```
#include<iostream.h>
main()
{ int n, x, d, s[30]={0};
  cin>>n;
  int i=0, nd=0;
// nd съдържа дължината на двоичното представяне на сумата
// въвеждат се числата
  for (int j=0;j<n;j++)
  {
      cin>>x; i=0;
//преобразуват се в двоичен вид и всяка от получените двоични
цифри се добавя към съответния елемент на сумата
      while (x>0)
      {
            d=x%2;x/=2; s[i]=d xor s[i]; i++;
      }
//обновява се дължината на получената сума
      if (i-1>nd)nd=i-1;
```

```
  }
//двоичната сума се преобразува в десетичен вид и се извежда
  x=0;
  for (int j=nd;j>=0; j--) x=x*2+s[j];
  cout<<x<<endl;
}
```

Решение с използване на побитови операции:
По условие двоичната сума се получава чрез събиране на цифрите от двоичното представяне по модул 2. Това позволява да използваме операция xor за намиране на двоичната сума. Първото от числата въвеждаме директно в s, където ще натрупваме сумата. Останалите числа въвеждаме последователно в цикъл и ги добавяме към сумата чрез xor. Така получаваме сумата в s, която в края извеждаме.

```
#include<iostream.h>
main()
{ int s, x, n;
  cin>>n;
  cin>>s;
  for (int i=1;i<n;i++)
  {
      cin>>x; s=s xor x;
   }
  cout<<s<<endl;
}
}
```

Задача D1. САМОЛЕТ

Самолет излита в K часа и M минути и пристига в L часа и N минути. Да се напише програма **PLANE,** която намира колко часа и колко минути е летял самолетът и кое негово време (излитането или кацането) е по-напред в денонощието си. Продължителността на полета е по-малка от 24 часа.

Излитането и кацането са в рамките на една и съща часова зона.

Вход:

Четири цели числа на един ред разделени с интервал K M L N ($0 \le K, L \le 23$, $0 \le M, N \le 59$).

Изход: Две цели числа за часове и минути за продължителността на полета на първия ред и "I" или "K" на втория ред в зависимост, от това кое е по-напред в денонощието си – излитането или кацането.

Примери:

	Пример 1	Пример 2	Пример 3
Вход	1 20 4 40	20 30 2 5	22 0 2 0
Изход	3 20 I	5 35 K	4 0 K

Решение:

При решаването на тази задача е удобно часовете и минутите да се превърнат в минути – по този начин се отчита само в една мярна единица времето изтекло от началото на съответното денонощие до излитането (изчислено е в променливата t1) и до кацането (изчислено е в променливата t2). Проверката на това кое е по-напред в денонощието след това е лесна – ако t1<t2, то тъй като полетът има продължителност по-малка от 24 часа те са в едно и също денонощие, а в противен случай в различни денонощия.

```
#include <iostream>
using namespace std;
int main(){
    int K, M, L, N, t1,t2, h, m;
    cin >>K >>M >>L >>N;
    t1=K*60+M;
    t2=L*60+N;
    if () {
        h=(t2-t1)/60;
        m=(t2-t1)%60;
        cout << h<<" "<<m<<endl<<"I\n";}
    else {
         h=(24*60+t2-t1)/60;
         m=(24*60+t2-t1)%60;
         cout << h<<" "<<m<<endl<<"K\n";}
    return 0;
    }
```

Задача D2. КАРТИНКА

Иванчо сега се учи да чете и пише на английски. По време на пътуване измислил една игра – преписва една дума от речника си на лист само с малки букви и след това я прави на картинка.

- За някои думи постъпва така: на всеки следващ ред преписва отново думата, но без последната буква от предходният ред и това се повтаря докато свърши думата. Например за “book”, се получава:

```
book
boo
bo
b
```

- За други – на следващия ред преписва горната дума, но без първата и последната буква, като запазва местоположението на буквите една под друга. И това се повтаря докато остане 1 или 2 букви, а после ги добавя постепенно отново. Например за “alabala”, се получава:

```
alabala
 labal
  aba
   b
  aba
 labal
alabala
```

Един ден той открил, че пише думите картинки винаги по втория начин, когато те могат да се четат по един и същ начин, от ляво на дясно и от дясно наляво. Кака му (от D група), нали си е винаги всезнайка, му обяснила, че такива думи се наричат “палиндроми” (Ух – отвратително е колко много знае :( )

Напишете програма **HOURGLASS**, която за дадена дума извежда картинката, която би нарисувал Иванчо.

Вход: Дума имаща не повече от 20 символа. Всички символи са английски букви.

Изход: Картинката, която би написал Иванчо.

Примери:

	Пример 1	**Пример 2**	**Пример 3**
Вход	Book	alabala	AbCcba
Изход	Book boo bo b	alabala labal aba b aba labal alabala	abccba bccb cc bccb abccba

Решение:

В задачата има три етапа:

1) Следният текст **“**преписва една дума от речника си на лист само с малки букви” означава, че текста трябва да се преобразува само в малки букви преди да се обработва.

2) Проверка за палиндром – в решението се използва променливата f за флаг, дали текста е палиндром или не (променливата f има стойност 1, ако не е палиндром, и 0 ако е палиндром)

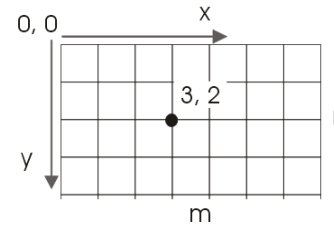
3) извеждането на необходимата фигура. При даденото от авторите решение при извеждането на фигурата се отпечатват интервали на местата, на които Иванчо не е писал съответната буква.

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
string d,s;
int i,f=0,l;
cin >>d;
l=d.length();
// 1) преобразуване в малки букви вариант 1
for (i=0; i<l; i++)
    if ( 'A' <= d[i] && d[i]<='Z') d[i]=d[i]+('q'-'Q'); //1
// по-елегантно е преобразуването в малки букви,
// с използването на функция tolower()
//вместо ред 1 се записва d[i] = tolower(d[i]);
//проверка за палиндром
for (i=0;i<l/2;i++)
    if (d[i]!=d[l-i-1]) {f=1; break;}
                          //намерени 2 съответни различни
                          // символа => не е палиндром
if (f) //ако не е палиндром -> триъгълник
   for (i=l-1; i>=0; i--) {
       cout <<d<< endl;
       d[i]=' ';
   }
else { //ако е палиндром -> пясъчен часовник
     s=d;
     l--;
     for (i=0; i<l/2; i++) {
         cout << d <<endl;
         d[i]=d[l-i]=' ';
     }
```

```
      for (i=l/2; i>=0; i--) {
          d[i]=d[l-i]=s[i];
          cout <<d <<endl;
      }
}
return 0;
}
```

Задача D3. ОТБОРИ

Една моментна снимка от сателит показва разположението на играчите на терена в една нова игра „CF”. В тази игра има много играчи от различни отбори, които са с различни цветове екипи и също такива по цвят каски. Тази снимка на игралното поле е съхранена в “графичен” файл. Играч на отбор, който поддържате се намира с топката в долния десен ъгъл на терена. Вашата задача е да напишете програма **CF**, която намира колко отбора участват в играта. Ако играч от друг отбор е застрашил любимеца ви и се намира до него вляво изведете цвета му. Ако там се намира играч от същия отбор изведете 0, ако няма никакъв друг играч точно вляво изведете -1.

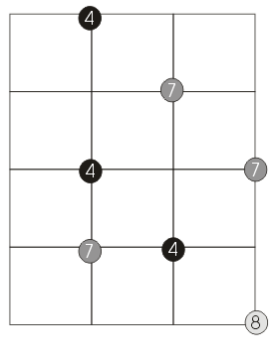
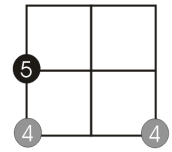
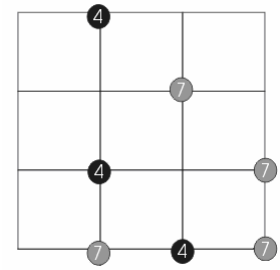


Един играч е най-близко вляво до любимеца ви, ако те имат една и съща координата по y и са на най-малко разстояние един от друг. Сигурно е, че никои двама играчи не се намират в този момент на едно и също място.

Вход: Въвежда се разположението на играчите на игралното поле.
На първия ред N (N<100) – брой играчи без любимеца ви.
На втория ред размерите на игрището m и n ($1 \le m < 640$, $1 \le n < 480$).
На всеки следващ ред за всеки играч: се въвеждат по три цели числа мястото му по x, по y и цвета му c ($0 \le x \le m$, $0 \le y \le n$, $0 \le c \le 255$), отделени с един интервал.
На последния ред цвят на любимеца ви.

Изход: На първия ред се извежда число, показващо броя на отборите, които участват в играта. На втория ред се извежда едно число -1, 0 или цвят, показващо вида на играча най-близко до любимеца ви вляво.

Примери

	Пример 1		Пример 2		Пример 3	
Вход	6 3 4 1 0 4 1 2 4 1 3 7 2 1 7 2 3 4 3 2 7 8		2 2 2 0 1 5 0 2 4 4		6 3 3 1 0 4 1 2 4 1 3 7 2 1 7 2 3 4 3 2 7 7	
Изход	3 -1		2 0		2 4	

Решение:

Първи начин:

Задачата може да се реши с три едномерни масива – за x, y и цвят c.
Различните отбори се съхраняват в масив otb, като променливата k съхранява колко елемента има попълнени в масива. При всеки нов играч се проверява: има ли го неговият отбор в масива otb и ако е необходимо се добавя. Не трябва да се забравя това действие да се направи и с "любимеца".
За втората част от задачата (намиране на съседа вляво) се обхожда отново масива y и за всеки елемент равен на m, се проверява дали той се среща за пръв път или не (флагът f). Динамично се съхранява най-близкият отляво (пази се за него x-са му в променливата leftx и цвета му в променливата leftc).

```
#include <iostream>

using namespace std;

int main (){
    int N, m, n, i, cl, j;
    cin >> N;
    cin >> m >> n;
    int x[N+1], y[N+1], c[N+1], otb[N+1];
    int k=0, f=0, leftx, leftc;
//въвеждане играчите
    for (i=0; i<N; i++) {
        cin >> x[i] >> y[i] >> c[i];
// проверка има ли такъв отбор в масива otb
        for (j=0; j<k; j++)
            if (otb[j]==c[i]) break;
// записване на отбора в масива otb, ако до момента не е в масива
        if (j==k) otb[k++]=c[i];
    }
    cin >> cl;
// проверка има ли го отбора на любимеца в масива otb
    for (j=0; j<k; j++)
        if (otb[j]==cl) break;
// записване на отбора в масива otb, ако до момента не е в масива
    if (j==k) otb[k++]=c[i];
// извеждане на броя отбори
    cout << k << endl;
// търсене на играч отляво
    for (i=0; i<N; i++)
        if (y[i]==n)
           if (f==0) {
              f=1;
              leftx=x[i];
              leftc=c[i];
           }
           else if (leftx<x[i]) {
                   leftx=x[i];
                   leftc=c[i];
                }
    if (f==0) cout << -1 <<endl;
    else if (leftc==cl) cout << 0 << endl;
         else cout <<leftc <<endl;
    return 0;
}
```

Втори начин:
Данните за всеки от играчите се съхраняват в масива t – представляващ полето за игра. Ако няма играч на дадени координати, то стойността на съответната клетка от t е -2, а ако има играч, то стойността на t е номера на цвета на екипа му. Преброяването на отборите е по-лесно, ако се използва структурата множество, в която поставяме различните цветове.

Намирането на играча отляво може да се получи с обхождане на n-тия ред, от играча в ъгъла - наляво. Ако се намери клетка със стойност различна от -2, то сме открили играч отляво и може да се изведе подходящата стойност – 0, ако е от същият отбор, или цвета на противника. Ако целият ред до началото има нулеви стойности, то никой не заплашва любимеца ни отляво.

```
#include <iostream>
#include <set>
using namespace std;
int main()
{    int N, k, i, m, n, j, col, fl=0;
     //в началото игрището е празно
     int t[640][480];
     for (i=0; i<640; i++)
          for (j=0; j<480;j++)
               t[i][j] = -2;
     //дефиниране на структурата множество
     typedef set<int,greater<int> > IntSet;
     IntSet set1;
     //четене на входните данни
     cin >> N;
     cin >> m >> n;
     for (k = 0; k < N; k++)
      {
      cin >> i;
      cin >> j;
      cin >> t[i][j];
      set1.insert(t[i][j]);
      }
     cin >> t[m][n];
     set1.insert(t[m][n]);
     //извеждане на броят на отборите
     cout <<  set1.size() << endl;
     //намиране на първият отляво
     for (i=m-1; i>=0; i--)
      if (t[i][n]!=-2)
        if (t[i][n] == t[m][n]) {cout << 0; fl=1; break;}
          else {cout << t[i][n]; fl=1;break;}
     if (!fl) cout << - 1;// не е намерил никого вляво
     cout << endl;
     return 0;
}
```

Задача Е1 . КОДИРАНЕ

Мая и Мира са ученички в четвърти клас и имат да си казват много тайни неща. Често те си разменят бележки, но се страхуват, че някой може да ги прочете, затова си измислили таен код. Всяка дума от бележката се пише с английски букви и се кодира, като всяка буква от нея се замества със следващата я в английската азбука, като буквата "z" се замества с "a". Всички букви в бележката са малки. Оказало се, че след като са кодирали текста им е трудно да го разчетат. За щастие открили, че в бележката има само четирибуквени думи.

Помогнете на двете момичета, като напишете програма **KOD**, която чете от клавиатурата дума, съставена от четири малки латински букви и отпечатва думата, чийто код е дадената.

ПРИМЕР 1		**ПРИМЕР 2**	
Вход	**Изход**	**Вход**	**Изход**
wpeb	voda	ajnb	zima

Решение:

В програмите на езика C++ буквите се представят чрез променливи от знаковия тип `char`. За запазването на четирибуквена дума са необходими четири знакови променливи.

```
char c1, c2, c3, c4;
```

За да прочетем думата, ще използваме **`cin.get()`** за всяка една от буквите.

```
c1=cin.get();
c2=cin.get();
c3=cin.get();
c4=cin.get();
```

Тъй като по условие е дадена кодираната дума, то всяка буква трябва да бъде заменена с предхождащата я в английската азбука. Изключение прави буквата **a**, която трябва да бъде заменена със **z**.

```
if(c1=='a')c1='z'; else c1++;
```

Така, че за всяка от въведените букви ще трябва да извършим преобразуването и след това да ги отпечатаме.

Ето решението на задачата:

```
#include<iostream>
using namespace std;
int main()
{
     char c1, c2, c3, c4;
     c1=cin.get();
     c2=cin.get();
     c3=cin.get();
     c4=cin.get();
     if(c1=='a')c1='z'; else c1++;
     if(c2=='a')c2='z'; else c2++;
     if(c3=='a')c3='z'; else c3++;
     if(c4=='a')c4='z'; else c4++;
     cout<<c1<<c2<<c3<<c4;
 }
```

Задача Е2. БИКОВЕ И КРАВИ

Всеки ден в час Мартин и Иван тайно играят на "Бикове и крави". Мартин си измисля четирицифрено число А, а Иван се опитва да го познае като предлага свое четирицифрено число В. Цифрите в числото не могат да се повтарят и числото не може да започва с цифрата нула. Когато числото на Иван съдържа цифра, която се намира на една и съща позиция в числото А и в числото В, Мартин му казва, че има "бик", а когато цифрата от числото В не е на същата позиция в числото А, Мартин му казва, че има "крава". Целта е Иван да познае числото на Мартин с минимален брой предложения. След това си разменят ролите и играта започва отначало. Победител е този от двамата, който е познал числото на другия с по-малък брой предложения.

Тъй като двамата не искат учителката им да разбере, че играят в час, те искат да напишат програма, която да намира броя на биковете и броя на кравите на две четирицифрени числа и да играят играта в час по информационни технологии с помощта на компютър. Като пораснат още малко ще се опитат да напишат и цялата игра на компютър. Но двамата не внимават много и в школите по информатика, затова трябва да им помогнете, като напишете програма **BULLCOW**, която прочита от клавиатурата две четирицифрени числа и извежда на екрана броя на "биковете" и броя на "кравите", разделени с един интервал.

ПРИМЕР 1		**ПРИМЕР 2**	
Вход	**Изход**	**Вход**	**Изход**
1234 3245	1 2	3056 7841	0 0

Решение:

АЛГОРИТЪМ:

В осем променливи ще се запишат цифрите на двете числа А и В, след което ще се сравняват. Ще са ни необходими и две цели променливи, в които ще натрупваме броя на биковете и броя на кравите(броя4 на биковете и брояч на кравите). Не забравяйте, че преди всичко на двата брояча трябва да присвоим стойност 0, за да могат правилно да отчитат. Нека цифрите на числото А са a1, a2, a3 и a4, а цифрите на числото В са b1, b2, b3 и b4. Тогава "бик" между двете числа има само ако са равни две съответни цифри от числото А и числото В (цифрите на единиците, на десетиците, на стотиците или на хилядните) – например a1 е равна на b1. Това се проверява с четири условни оператора, като всеки път, когато условието за равенство е вярно се увеличава с 1 броячът на "биковете".

По подобен начин се намира и броят на "кравите". Но тук се увеличава с 1 брояча на кравите, когато една от цифрите на числото А е равна на някоя от цифрите на числото В, която **не е** на същата позиция – например a1 е равна на b2 или a1 е равна на b3 или a1 е равна на b4. Това отново се проверява с четири условни оператори.

Намирането на цифрите на двете числа се извършва с операциите за целочислено деление и деление с остатък на 10.

ПРОГРАМА (на език C++)

```
#include<iostream.h>
int main()
{
     long a, b;
     int a1, a2, a3, a4, b1, b2, b3, b4, brbull, brcow;

     cin>>a>>b;
     brbull=0;
     brcow=0;
```

```
        //namira cifrite na chisloto A
        a1=a%10;
        a2=(a/10)%10;
        a3=(a/100)%10;
        a4=(a/1000)%10;
        //namira cifrite na chisloto B
        b1=b%10;
        b2=(b/10)%10;
        b3=(b/100)%10;
        b4=(b/1000)%10;
        //namira broq na bikovete
        if (a1==b1) brbull++;
        if (a2==b2) brbull++;
        if (a3==b3) brbull++;
        if (a4==b4) brbull++;
        //namira broq na bikovete
        if (a1==b2 || a1==b3 || a1==b4) brcow++;
        if (a2==b1 || a2==b3 || a2==b4) brcow++;
        if (a3==b1 || a3==b2 || a3==b4) brcow++;
        if (a4==b1 || a3==b2 || a3==b3) brcow++;
        cout<<brbull<<" "<<brcow<<"\n";
        return 0;
}
```

Задача Е3. ТРИЪГЪЛНИК

Един ден Пешо намерил в училищния двор три прави пръчки и решил да направи от тях триъгълник, така че пръчките да са страни на триъгълника, но не успял.

Вечерта като се прибрал в къщи, Пешо разказал случката на баща си и той му обяснил, че не от всеки три отсечки може да се образува триъгълник. Трябва дължината на всяка от отсечките да е по-малка от сбора на дължините на другите две.

Напишете програма **TRIANG**, която прочита от клавиатурата дължините на три отсечки и извежда на екрана думата "NO", ако не може да се направи триъгълник със страни тези отсечки. Ако може да се направи триъгълник, да се изведе периметъра на триъгълника и числото 1, ако той е равностранен, 2 – ако е равнобедрен и 3 – ако е разностранен. Двете числа се извеждат на един ред, разделени с интервал.

ПРИМЕР 1		**ПРИМЕР 2**	
Вход	**Изход**	**Вход**	**Изход**
5 16 3	NO	3 4 5	12 3
ПРИМЕР 3		**ПРИМЕР 4**	
Вход	**Изход**	**Вход**	**Изход**
10 10 10	30 1	12 4 12	28 2

Решение:

АЛГОРИТЪМ:

Въвеждат се три числа, които се записват в три цели променливи – например a, b, и c. След това първо се проверява дали съществува триъгълник с дължини на страните равни на тези числа, т.е. дали всяка от страните е по-малка от сборът на останалите две. Това се прави с условен оператор, който съдържа три условия съединени с връзката И. Ако условието е вярно, се пресмята периметъра на триъгълника и се извежда, заедно с интервала след него, след което се прави проверка дали той е равностранен (три равни страни), равнобедрен (точно две равни страни) или разностранен (три различни страни) Ако условието не е вярно, т.е. не съществува триъгълник, се извежда думата “NO”. Проверките за вида на триъгълника се правят с вложени условни оператори, което спестява проверка на някои от условията – например, ако триъгълникът не е равностранен, той е или равнобедрен или разностранен, а ако не е и равнобедрен, то той е разностранен.

ПРОГРАМА:

```
#include<iostream.h>
int main()
{
     long a, b, c, p;
     cin>>a>>b>>c;
     p=a+b+c;
     if (a<b+c && b<a+c && c<a+b)
     {
          cout<<p<<” “;
                         if (a==b && b==c) cout<<1<<”\n”;
          else
          if (a==b || b==c || a==c) cout<<2<<”\n”;
          else cout<<3<<”\n”;
     }
     else
          cout<<”NO”<<”\n”;
     return 0;
}
```