Задача B1. Две и три

Разполагаме с неограничен брой десетични цифри 2 и 3. С част от тях искаме да напишем десетично число, което да се дели на 2^n, където n е естествено число. При това търсим най-малкото такова число.

Ако например $n = 4$, една цифра не стига да напишем такова число: 2 е четно, но не се дели на $2^4 = 16$, 3 пък направо е нечетно. С две цифри от наличните могат да се напишат (по големина) 22, 23, 32 и 33. Числото 32 вече има нужното ни свойство (впрочем, то се дели даже и на $2^5 = 32$). Факт е, че например 33232 също изпълнява условието, но то е много по-голямо. Няма изискване и двете цифри да се срещат задължително в записа, нито пък да са с равен брой срещания.

Напишете програма **N23**, която намира желаното число или установява, че такова не съществува.

От стандартния вход се въвежда един ред, съдържащ естественото число n, $1 \le n \le 10000$.

На стандартния изход програмата трябва да изведе един ред с намереното най-малко естествено число, в десетичния запис на което има само цифрите 2 и 3 (по колкото е необходимо от всяка от тях) и което се дели на 2^n; или един ред със съобщението NO, ако такова число не съществува.

Пример. Вход:
```
11
```

Изход:
```
223232
```

Решение

Още от основния училищен курс за десетично записаното естествено число m е известно, че:

- дели се на 2, ако е четно (т.е. – последната му (*една*) цифра се дели на 2);
- дели се на 4, ако числото, образувано от последните му *две* цифри се дели на 4 (ако е едноцифрено, можем да считаме, че има водеща нула);
- дели се на 8, ако числото, образувано от последните му *три* цифри се дели на 8 (отново можем да дописваме водещи нули, ако няма достатъчно цифри).

Обобщението на тези правила се очаква и лесно се доказва: $\boldsymbol{m}$ се дели на 2^n тогава и само тогава, когато числото, образувано от последните му $\boldsymbol{n}$ цифри в същия ред се дели на 2^n (ако цифрите на $\boldsymbol{m}$ не стигат, можем да си мислим, че има водещи нули).

Като вземем предвид и елементарното съображение, че щом $\boldsymbol{m}$ се дели на 2^n, то се дели и на всички по-малки степени на двойката, заключаваме, че с ограничения ресурс от десетични цифри трябва да изграждаме „опашки" от $\boldsymbol{n}$ цифри, гарантиращи делимостта на 2^n. В нашия случай се оказва, впрочем, че такава „опашка" е еднозначно определена. Наистина – последната цифра задължително е 2, иначе пропада делимостта на 2; предпоследната – задължително 3, което осигурява делимост на 4; следващата по старшинство – задължително 2, иначе няма делимост на 8 и т. н.

Алгоритъмът за изграждане на тази редица се оказва елементарен, което еднозначно ни осигурява число, записано с общо n на брой цифри (двойки и тройки), което се дели на 2^n, т.е. задачата винаги има решение с не повече от n цифри. По-интересно е изискването да се намери най-малкото решение. Наистина, 23232 (което е опашката от пет цифри) се дели на 2^5=32, но още 32=00032 има това свойство (и се записва само с две цифри).

Алгоритъм за получаване на опашката. Да разгледаме сега по-подробно аритметиката на получаване на редицата $\{\boldsymbol{c_i}\}$ от цифри в „опашката", започвайки от най-младшата – $\boldsymbol{c_1}$. Единствената възможност, както споменахме, е $\boldsymbol{c_1}$=2. Всяка от следващите (по старшинство) цифри от „опашката" m трябва да осигурява делимост на още една степен на двойката. На i-тата стъпка ($i > 1$) ще имаме $\boldsymbol{c_i}$=2 или $\boldsymbol{c_i}$=3, т. е., ако означим с

$$m_i = \overline{c_i \underbrace{c_{i-1} c_{i-2} \dots c_1}_{m_{i-1}}}$$

естественото число, записано с тези цифри в този ред, имаме:

$$m_i = \begin{cases} 2.10^{i-1} + m_{i-1} \\ \textit{или} \\ 3.10^{i-1} + m_{i-1} \end{cases}$$

като изборът е еднозначно определен от това, дали „досегашната опашка" m_{i-1} вече се дели и на 2^i (на 2^{i-1} тя задължително се дели) или не. Наистина, ако $m_{i-1} = 2^i.k$, където $\boldsymbol{k}$ е естествено число, единствената възможност да получим още един делител на 2 е да вземем първата алтернатива:

$$m_i = 3.10^{i-1} + m_{i-1} = 3.2^{i-1}.5^{i-1} + 2^{i-1}.t = 2^{i-1}(3.5^{i-1} + t)$$

(не втората, защото тогава първото събираемо не се дели на 2^i, второто се дели → сумата не се дели). И точно напротив – при $m_{i-1} \neq 2^i.k$ ще имаме $m_{i-1} = 2^{i-1}.t$, където $\boldsymbol{t}$ е нечетно. Единствена възможност сега остава

$$m_i = 2.10^{i-1} + m_{i-1} = 2.2^{i-1}.5^{i-1} + 2^i.k = 2^i(5^{i-1} + k)$$

като числото в скобите е четно (сума от две нечетни) и, следователно, произведението се дели на 2^i. По аналогични на предишния случай причини другата алтернатива не може да бъде избрана. Това доказва еднозначността и показва алгоритъма за получаване на редицата от цифри $\{\boldsymbol{c_i}\}$.

Алгоритъм за решаване на задачата. Въз основа на горните разглеждания, естественият подход към атакуване на проблема е да изграждаме опашката по указания алгоритъм и да спрем в първия момент, когато получим делимост на 2^n. Малко запомняне на предишни резултати ще ускори изчислителния процес.

1. Въвежда се естественото число n.
2. Инициализираме променливи: резултат: res = 2; степен на петицата: deg5 = 1; изчислена база за пресмятане на новата стойност k = 1; осигурена степен на двойката, на която res се дели: deg = 1.
3. Проверяваме дали deg е по-малко от n. Ако да – към т. 4 иначе към т. 11.
4. Ако k е четно, долепваме към res водеща цифра 2, иначе – водеща цифра 3.
5. Минимално осигурената степен на делимост сега е равна на броя на цифрите на res, затова deg приема тази стойност.
6. Проверяваме дали deg все още е по-малко от n. Ако да – към т. 7 иначе към т. 11
7. Намираме следващата степен на петицата: deg5 := 5*deg5
8. k :=((новодобавената цифра)*deg5+k) / 2
9. Проверяваме каква степен на двойката дели k, като увеличаваме deg с толкова. Ако достигнем (или надхвърлим) **n**, към т. 11.
10. Към т. 4.
11. Извеждаме резултата res.

Примерна реализация

```
#include <stdio.h>
int n;
typedef struct {int count;
                char dig[10001];
               } Long;

Long res={1,{2}},deg5={1,{1}},k={1,{1}};

void Long_Add(Long *a, Long *b, Long *r)
{char carry=0;
 int i;
 r->count=(a->count>b->count)?a->count:b->count;
 for (i=0;i<r->count;i++)
 {if (i<a->count&&i<b->count) r->dig[i]=a->dig[i]+b->dig[i];
  else r->dig[i]=(i<a->count)?a->dig[i]:b->dig[i];
  r->dig[i]+=carry;
  if (r->dig[i]>9) {r->dig[i]-=10;carry=1;}
  else carry=0;
 }
 if (carry) r->dig[r->count++]=1;
}
```

```
void Long_MulDig(Long *a,char d,Long *r)
{char carry=0;
 int i,t;
 r->count=a->count;
 for(i=0;i<a->count;i++)
 {t=d*a->dig[i]+carry;
  r->dig[i]=t%10;
  carry=t/10;
 }
 if (carry) r->dig[r->count++]=carry;
}

int Long_Div2(Long *a,Long *r)
{int i,d=0;
 r->count=a->count;
 for (i=a->count-1;i>=0;i--)
 {d=10*d+a->dig[i];
  r->dig[i]=d>>1;
  d&=1;
 }
 if (!r->dig[r->count-1]) r->count--;
 return d;
}

void Long_Show(Long *a)
{int i;
 for (i=a->count-1;i>=0;i--) printf("%d",a->dig[i]);
 printf("\n");
}

int main(void)
{int deg=1;
 Long t;
 scanf("%d",&n);
 while (deg<n)
 {res.dig[res.count++]=2+(k.dig[0]&1);
  deg=res.count;
  if (deg==n) break;
  Long_MulDig(&deg5,5,&deg5);
  Long_MulDig(&deg5,res.dig[res.count-1],&t);
  Long_Add(&t,&k,&t);
  Long_Div2(&t,&k);
  t=k;
  while (deg<n&&!Long_Div2(&t,&t)) deg++;
 }
 Long_Show(&res);
 return 0;
}
```

Задача B2. Папки

Хакер Ш. имал два компютъра – A и B. На твърдия диск на A той работел в папка с дърво от подпапки в нея, където били записани файловете му. От време на време Ш. преписвал на твърдия диск на B цялата своя папка от A и така правел back-up. На компютъра B нищо друго не се правело. Но с течение на времето файловете му ставали все повече и той решил да копира само тези от тях, които са по-нови в сравнение с вече копираните от предишния път. Разбира се, ако е направил нова подпапка, копирал я цялата, заедно с всичките нейни подпапки и файлове.

Напишете програма **FOLDERS**, която при зададено описание на папките в A и B, пресмята броя на файловете, които трябва да се копират.

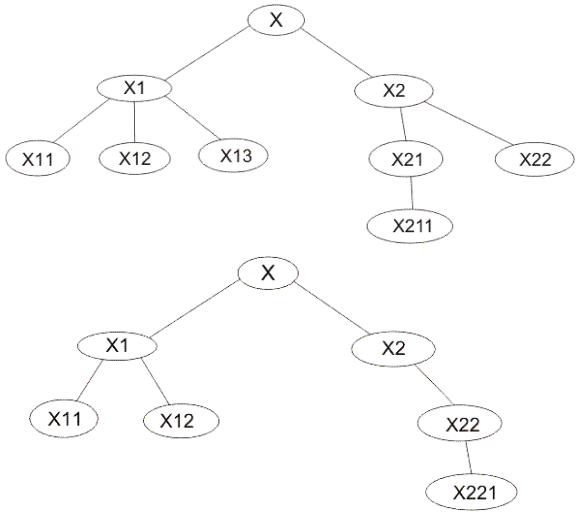
Входните данни се четат от стандартния вход. Дървовидната структура на папките е зададена, започвайки с главната папка, чието име е написано в първия ред на входния файл (низ от малки латински букви и цифри), слевано от броя на намиращите се в нея подпапки и/или файлове и времето на създаването й (в секунди, от някакъв начален нулев момент). На следващите редове са нейните подпапки и файлове, като в случая, когато е даден файл, броят на съдържащите се в него подпапки и файлове е отбелязан с 0, а даденото време се отнася за момента на последната промяна във файла. Изобщо казано, входният файл е образуван, като първо е записан коренът на дървото, след него – неговите наследници. След това, след всеки наследник се вмъкват редове за неговите наследници (ако има такива) и така нататък, докато се изчерпят всички наследници. След като се запише файловата структура на A, във входния файл се записва по аналогичен начин файловата структура на B.

Програмата трябва да изведе броя на файловете, които трябва да бъдат копирани от A в B.

Пояснения и ограничения: Няма еднакви имена в една папка. Няма празни папки. Ако две папки (или файлове), едната намираща се в A, а другата – в B, са с еднакви имена и са на едно и също съответно място в йерархията, но папката (файлът) в A е създадена по-късно от тази в B, копира се цялата й система от подпапки и файлове (ако е файл – копира се само файлът, като той заменя съответната папка в B, която се изтрива). Максимален брой редове във входния файл: 200. Максимален брой подпапки и/или файлове в една папка: 9. Максимална дължина на името на файл или папка: 9 Максимален момент от времето 9999 сек.

Пример. Вход:

```
x 2 0
x1 3 1
x11 0 14
x12 0 11
x13 0 13
x2 2 5
x21 1 20
x211 0 21
x22 0 16
x 2 0
x1 2 1
x11 0 9
x12 0 11
x2 1 5
x22 1 6
x221 0 7
```



Изход:

```
4
```

Решение:

Структурата данни, в която се пази дървото на папките от А използва масивите `int a[N][M], at[N]; char as[N][10];` В `at[j]` е записан съответният момент време за `j`-тата папка (или файл), в `as[j]` – името й. Броят на наследниците й се съхранява в елемента `a[j][0]`, а номерата на наследниците – в `a[j][1]`, `a[j][2]`, ..., `a[j][a[j][0]]`. В `na` е общият брой на всички папки и файлове от А. По аналогичен начин се изгражда структурата от данни за В, като се ползват съответно `b[N][M], bt[N], bs[N][10]` и `nb`. Запълването на тези данни се извърша от фукцията create(), извиквана два пъти, за да обработи входния файл за А и за В.

Функцията `compare(int n)` рекурсивно и синхронно обхожда двете дървовоидни структури, спускайки се в дълбочина. Всеки път, когато се достигне връх на дървото А, при което се открива, че съответния връх на В няма необходимите свойства за "еднаквост" (например върхът в А е папка, а този в В – е файл, или двата върха са файлове, но този в А е с по-ново време), навлизането в дълбочина се прекратява и евентуално се извиква рекурсивната функция `count()`, която преброява файловете от поддървото на А, което започва от въпросния връх. Променливата `c` служи за глобален брояч, чиято стойност се отпечатва накрая на програмата. Във функцията `main()` се обработва коренът на дървото, който има номер 1.

```
//ANSI C++
#include<iostream>
#include<cstring>
using namespace std;

const int N=999;
const int M=19;
int a[N][M], at[N]; char as[N][10];
int b[N][M], bt[N]; char bs[N][10];
int n,na,nb;
int c=0;

void create(int a[N][M], int at[N], char as[N][10])
{
 cin >> as[n] >> a[n][0] >> at[n];
 int n0=n;
 for(int i=1; i<=a[n0][0]; i++)
  {
    n++;a[n0][i]=n;
    create(a, at, as);
  }
}

void count(int n)
{
 if(a[n][0]==0) c++;
 else
 for(int i=1;i<=a[n][0];i++) count(a[n][i]);
}

void compare(int n)
{
 int i,j,j0;
 for(i=1; i<=a[n][0]; i++)
  {
    int f=0;
    for(j=1; j<=b[n][0]; j++)
```

```
     if(strcmp(as[a[n][i]],bs[b[n][j]])==0) {f=1; j0=j;}
    if(f==1)
    {
     if(at[a[n][i]]==bt[b[n][j0]])
      {
        if((a[a[n][i]][0]!=0)&&(b[b[n][j0]][0]!=0)) compare(a[n][i]);
        else if((a[a[n][i]][0]==0)&&(b[b[n][j0]][0]!=0)) c++;
        else
         if((a[a[n][i]][0]!=0)&&(b[b[n][j0]][0]==0)) count(a[n][i]);
      }
     if(at[a[n][i]]>bt[b[n][j0]]) count(a[n][i]);
    }
    if(f==0) count(a[n][i]);
  }
}

int main()
{
 n=1;create(a, at, as);na=n;
 n=1;create(b, bt, bs);nb=n;
 if(strcmp(as[1],bs[1]) != 0) count(1);
 else if((a[1][0]==0)&&(b[1][0]!=0)) count(1);
 else if((a[1][0]!=0)&&(b[1][0]==0)) count(1);
 else if(at[1] > bt[1]) count(1);
 else compare(1);
 cout << c << "\n";
 return 0;
}
```

Задача B3. Зелени площи

В град Ш. има много зелени площи. Всяка от зелените площи е с формата на изпъкнал многоъгълник. За да се поддържат зелените площи в изряден вид, кметът на града решил да помоли Директора на Математическата гимназия в Ш. да възложи на всеки ученик поддържането на поне една от тях. Директорът пък решил да възложи на ученика от 9 клас с профил Информатика Шибил да направи програма, която по случаен начин да разпредели площите между учениците. Шибил, обаче, решил да се възползва от възможността и да остави за себе си най-малката площ. Понеже току що е започнал да учи програмиране, задачата му се сторила много трудна. Помогнете му, като напишете програма **GREEN**, която да намира зелената площ с минимално лице.

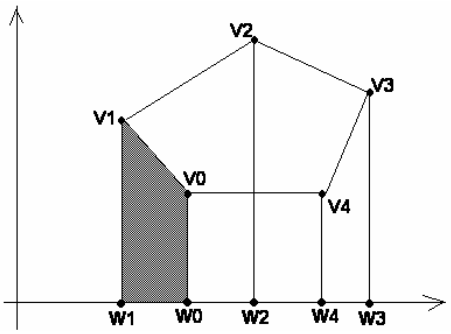
Входните данни ще бъдат зададени на стандартния вход. На всеки ред ще бъде зададено описанието на един от многоъгълниците. То започва с броя M на върховете на многоъгълника ($3 \le M \le 100$), последван от M-те двойки целочислени координати на върховете (в правоъгълна координатна система), зададени в посока на часовниковата стрелка. Всички числа са разделени с по един интервал. След описанието на последния многоъгълник следва ред, съдържащ само числото 0.

На стандартния изход програмата трябва да изведе номера на многоъгълника с най-малко лице, като номерацията на многоълниците започва от 1 и следва реда, по който те са зададени на входа. Ако два или повече многоъгълника са с едно и също минимално лице, да се изведе номера на този от тях, който има по-малко върхове. Ако два или повече многоъгълника са с едно и също минимално лице и един и същ брой върхове, да се изведе този от тях, който има по-малък номер.

Пример. Вход:
```
4 13 11 13 21 23 21 23 11
3 -15 -5 0 5 15 -5
4 -5 -5 -5 5 5 5 5 -5
0
```
Изход:
```
1
```

Решение:

Задачата за намиране на лице на многоъгълник е основна за изчислителната (комбинаторната) геометрия. Освен пряко по предназначение, намирането на лицето на многоъгълник може да се окаже важна стъпка при решаване на други интересни задачи. Преди да се спрем на авторското решение, нека да отбележим, че за намирането на лице на изпъкнал многоъгълник могат да се предложат различни алгоритми. Една възможност е да разбием многоъгълника с върхове V_0, V_1,…, V_{N-1} на триъгълници $V_0V_1V_2$, $V_0V_2V_3$,…, $V_0V_{N-2}V_{N-1}$, да намерим лицето на всеки триъгълник, например с използване на формулата на Херон и да пресметнем лицето на изпъкналия многоъгълник като сума от лицата на триъгълниците. Подобен подход има сериозен недостатък – изчисленията при такъв алгоритъм са свързани със загуба на точност. При формулата на Херон ще са ни необходими дължините на страните на триъгълника, при намирането на които ще се наложи използването на коренуване, а и при пресмятането на окончателния резултат ще се наложи да се коренува още веднъж. В изчислителната геометрия коренуването, както делението и пресмятането на функции като sin, cos, tg, arctg и т.н., често води до загуба на точност. Затова използването на такива пресмятания трябва да се избягва или да се сведе до минимум.



Фиг. 1

За решаване на задачата се предлага алгоритъм, който не предизвиква подобни изчислителни проблеми – т.н. *ориентирани лица*. Същността на алгоритъма е в следната формула:

$$S=|(x_1-x_0)*(y_1+y_0)/2+(x_2-x_1)*(y_2+y_1)/2+\ldots+(x_{N-1}-x_{N-2})*(y_{N-1}+y_{N-2})/2+(x_0-x_{N-1})*(y_0+y_{N-1})/2|$$

където (x_I,y_I), $I=0,1,\ldots,N-1$, са координатите на върховете на изпъкналия многоъгълник, в реда, в който се срещат при обхождането му (по посока на часовниковата стрелка или в обратна посока). На верността на тази формула няма да се спираме тук, а само ще отбележим, че стойността на израза $(x_I-x_{I-1})*(y_I+y_{I-1})/2$ е ориентираното лице на трапеца $V_{I-1}V_IW_IW_{I-1}$, където W_I и W_{I-1} са ортогоналните проекции на V_I и V_{I-1} върху абсцисната ос (вж. Фиг. 1 където е защриховано лицето на трапеца $V_0V_1W_1W_0$). Ориентирано лице е лицето на трапеца със знак плюс или минус в зависимост от това дали W_I е по-надясно от W_{I-1} върху абсцисната ос. Така ориентираното лице на защрихования на Фиг.1 трапец е отрицателно и при пресмятането ще бъде извадено от положителното ориентирано лице на трапеца $V_1V_2W_2W_1$.

Забележете още, че въпросната формула е валидна не само за изпъкнали многоъгълници, но и за неизпъкнали несамопресичащи се.

И така, за пресмятане на лицето на един многоъгълник не е необходимо да запомняме координатите на върховете му в паметта. Достатъчно е да помним само координатите на началната точка, за да пресметнем последното събираемо на израза. За намирането на минимума също не е необходимо да запомняме намерените лица.

На Фиг. 2 е показан програмен фрагмент с основните стъпки на алгоритъма.

```
minface=(double)MAX; minv=MAX; minn=MAX;
i=1;
while(1)
{
   cin>>M;
   if(M==0) break;
   face=0.;
   cin>>x1>>y1; x0=x1;y0=y1;
   for(j=2;j<=M;j++)
   {
     cin>>x2>>y2;
   face+=(x2-x1)*(y2+y1)/2;
   x1=x2;y1=y2;
   }
   face+=(x0-x1)*(y0+y1)/2;
   if(face<0) face= -face;
   if((face<minface)||(face==minface&&M<minv))
   {minface=face;minv=M;minn=i++;}
}
cout<<minn<<endl;
```

Фиг. 2