

**VII НАЦИОНАЛЕН
ЕСЕНЕН ТУРНИР
ПО ИНФОРМАТИКА
И ИНФОРМАЦИОННИ
ТЕХНОЛОГИИ
“ДЖОН АТАНАСОВ”**

***Секция “Информатика”
Условия и решения***

**ШУМЕН, 23-25 Ноември
2007**

Организатори:

Министерство на
образованието и науката
Шуменски университет
“Епископ Константин
Преславски”
Съюз на математиците в

България
Школа А&Б – Шумен

Спонсори:

Генерален спонсор:
Microsoft – България

Министерство на
образованието и науката

Шуменски университет
“Епископ Константин Преславски”

Съюз на математиците в България

Школа А&Б – Шумен

„Еленко Компютърс”

Задача А1. ХАМАЛИ

Боклукът, събран след пролетното почистване на града, е пакетирани в бали и трябва да бъде извозен до сметището. Тази нелека задача е възложена на хамалите от „Кърти, Чисти, Извозва” ООД. Те не са много заприжени за околната среда и решили да използват канализацията на града за да транспортират балите. Канализацията има N шахти, номерирани от 1 до N ($1 \leq N \leq 150\ 000$), като шахтата с номер 1 се намира на сметището. Всяка шахта U ($1 < U \leq M$) е свързана с тръба към точно една шахта $V < U$ и течението ще отнесе до V всеки предмет, пуснат в U . Така всяка бала с боклук, пусната в U може да стигне до всяка шахта, която се намира на пътя от U до 1. стига в шахтата да има достатъчно вода, за да не заседне балата в нея.

В 10.00:00 сутринта, когато започва пренасянето на боклука, всички шахти са празни. За радост на хамалите, в този момент започва да вали дъжд и шахтите започват да се пълнят с 10 куб.см вода в секунда. Затова хамалите използват следната тактика. Един хвърля бала боклук в шахта $X(1 < X)$, а друг я чака в шахта Y ($Y < X$), разположена по пътя от X до 1. Максималният обем на бала, която може да бъде пренесена така, е равен на минималното количество вода в куб.см., което се съдържа в шахта по пътя от X до Y в момента на пускането на балата и това е което вълнува хамалите при всяка такава операция. Отговорът на въпроса ще се да бъде тривиален, ако конкурентите от „Цепи-Лепи” АД, не изпомпваха вода от някои от шахтите. Нاپишете програма **porters** за да им помогнете.

Вход.

На първия ред на стандартния входа са зададени числата N и $K(1 \leq N \leq 150\ 000)$. На втория ред са записани $N-1$ числа, като числото в позиция P е най-много P и означава номера на шахта, с която е свързана шахтата $P+1$. Всеки от следващите K реда е от един от следните два типа:

$k4i < T > < X > < Y >$, където $0 \leq T \leq 1\ 000\ 000$, $1 < X \leq N$, $1 \leq Y < X$ и Y е на пътя от X до 1. Това означава, че хамалите се интересуват от максималния обем на бала, която може да бъде пусната в X , T секунди след началото на извозването, и да стигне до Y ;

$серу_лепу < T > < X > < V >$, където $0 \leq T \leq 1\ 000\ 000$, $1 < X \leq N$, $0 \leq V < X$ и V е на пътя от X до 1. Това означава, че хамалите се интересуват от максималния обем на количеството вода в шахта X в момент T . Това означава, че конкурентите от „Цепи-Лепи” изпомпват V куб.см. вода от шахта X в момент, в който са изминали T секунди от началото на извозването. Изпомпването става моментално. Тези K реда са подредени в нарастващ ред на T и няма редове с еднаква стойност на T .

Изход.

За всеки ред от първи тип, срещнат във входа, програмата трябва да изведе на стандартния изход един ред с търсения максимален обем на бала. За ред от втори тип не се извежда нищо.

Примерен вход	Исход за примерния вход
8 7	0
1 1 3 3 2 4 3	20
k4i 0 5 3	30
k4i 2 8 3	40
seru_leru 5 6 60	
seru_leru 6 2 40	
seru_leru 7 1 50	
k4i 8 2 1	
k4i 10 6 1	

Решение:

Можем да перефразиране условието на задачата в термините на теорията на графите по следния начин: Дадено е кореново дърво, върховете на което са шахтите, номерирани с числата от 1 до M . Върхът с номер 1 е корен. Гръбноводите, свързващи всяка шахта (без тази с номер 1) с шахта с по-малък номер са ребрата на дървото. Да означим с $\tau(Z)$ бащата на Z , а с $\pi(X, Y)$ – пътя от X до Y в кореновото дърво.

На всеки връх v на дървото е съпоставено неотрицателно число $C(v, T)$ количеството дълждовна вода, в шахтата, представена от върха v . Това число зависи от времето T , изминало от началото и от направените изпомпвания. В началото $C(v, 0) = 0$ за всички върхове на дървото. След това $C(v, T)$ нараства пропорционално на изминалото време във всички върхове – с Q куб. см за всяка секунда (в условието Q е фиксирано на 10). В определени моменти от време $C(v, T)$ намалява със зададен обем, изпомпена от шахтата, вода.

Според условието на задачата, трябва да реализираме следните две операции:

1) По зададен момент T да се намери намери $\min\{C(v, T)\}$, където минимумът се взема по всички върхове v , принадлежащи на единствения път от връх A до връх B , като се знае, че връхът B принадлежи на пътя от A до корена на дървото (да наречем тази операция **query**).

2) По зададен момент T , връх v и число D , $0 < D \leq C(v, T)$, да се намали $C(v, T)$ с D (да наречем тази операция **update**).

За създаване на ефективен алгоритъм, трябва да се избере такова представяне на данните, че реализацията и на двете операции да е достатъчно бърза.

Нека за момент се абстрахираме от конкретната задача и разгледаме един подход за решаване на подобен тип задачи. Той се състои в разделяне на голямата задача на определен брой по-малки. Колкото по-малък е този брой, толкова по-бързо е изпълнението на операция **query** и толкова по-бавно е изпълнението на операция **update**. За ефективно решение трябва да се “балансира” броят.

Сега да се върнем към нашата задача. На всеки връх X да съпоставим:

- връх $X' = \text{Jump}(X)$, като $X' \in \pi(X, 1)$;

- число $F(X)$ – максималното количество изпомпана вода за $Y \in \pi(X, \text{Jump}(X))$;

- число $W(X)$ – общото количество вода, изпомпана от върха X .

Да отложим временно дискусиата за това, как точно ще избираме върха $\text{Jump}(X)$, как ще проверяваме дали $Z \in \pi(X, Y)$ и кои са върховете Y , за които $X \in \pi(Y, \text{Jump}(Y))$. Да разгледаме следните реализации на двете операции (използващи вече дефинираните стойности $\text{Jump}(X)$, за всеки връх X).

1) Операция **query** (T, A, B) – намиране на максималното количество изпомпана вода от някой връх, принадлежащ на пътя между A и B :

$C=A; X=0; //$ в X ще получим търсеното максимално количество

Докато ($C!=B$)

{ Ако ($\text{Jump}(C) \in \pi(C, B)$) { $X=\max(X, F(C))$; $C=\text{Jump}(C)$; }

Иначе { $X=\max(X, W(C))$; $C=\tau(C)$; }

}

Резултат $T*Q - X$;

2) Операция **update** (T, A, D)

$W(A)=W(A)+D$;

За всеки $X, A \in \pi(X, \text{Jump}(X))$

{ $F(X) = \max(F(X), W(A))$; }

Нека направим оценка на сложността на алгоритъма, като приемем че за всеки връх X така сме подбрали $\text{Jump}(X)$, че броят на върховете от $\pi(X, \text{Jump}(X))$ е най-много M (колкото може по-близо до M без да го надхвърля), а за всеки връх X броят на върховете X , такива че $Y \in \pi(X, \text{Jump}(X))$ е най-много M . Тогава операцията **query** ще бъде със сложност $O(N/M+M)$, защото цикълът обхожда един път в дървото с максимална дължина M , правейки, докато е възможно, стъпки с дължина около M (могат да останат не повече от M стъпки с дължина 1). Операцията **update**, пък, ще бъде със сложност $O(M)$, защото тялото на цикъла се изпълнява по един път за всеки връх X такъв, че $A \in \pi(X, \text{Jump}(X))$.

Сега сложността на алгоритъма ще бъде $p.O(N/M+M) + q.O(M)$, където p и q са броят на операциите **query** и **update**, съответно. Остава да изберем оптимална стойност за M . В случай че изберем $M=\sqrt{N}$, тогава двете операции ще бъдат със сложност $O(\sqrt{N})$ и ще получим за сложността на целия алгоритъм $O((p+q)*\sqrt{N})$. Не е невъзможно да се подходи още по-дълбоко – да се прочете целия входен файл, да се намерят точните стойности на p и q , и като се има предвид времеовото ограничение да се пресметне най-подходящата стойност на M по-точно.

Остава е да видим как, за всеки връх X да изберем $\text{Jump}(X)$ така, че да е изпълнено споменатото по-горе условие. Предлагаме следния алгоритъм:

/* С обхождане на дървото в постордер (бащата се обхожда след

като са обходени всички синове), започвайки от корена

пресмятаме стойностите **Weight** и **Depth** */

Depth (1)=1; **Weight** (A) = 1;

Остава да отбележим само, че проверката $\text{Jump}(C) \in \pi(C, B)$ извършваме много лесно, защото $\text{Jump}(C)$ и B са върхове по един и същ път до корена и значи условието е изпълнено когато $\text{Depth}(\text{Jump}(C)) > \text{Depth}(B)$. Колкото до определянето на върховете A , за които $X \in \pi(A, \text{Jump}(A))$, това може да направим по време на пресмятането на $\text{Jump}(A)$, където започваме от $\tau(A)$ и се движим по пътя към корена. При това трябва да запомним за всеки срещнат по този път връх X , че принадлежи на $\pi(A, \text{Jump}(A))$. В програмата това става в масива от вектори U .

```
#include <cstdio>
#include <cstring>
#include <stack>
#include <vector>
using namespace std;
#define MAXN 1<<18
#define SQRT 400
#define C_NAME "k4i" // QUERY_COMMAND_NAME
#define Q 10 // RAIN_WATER_PER_SEC
int N, Root[MAXN], Depth[MAXN], Weight[MAXN];
int Water[MAXN], Jump[MAXN], F[MAXN], NumberOfTasks;
vector<int> A[MAXN], U[MAXN];
void init();
void calcDepthAndWeight();
void buildJump(const int v);
void solveQuery();
void doUpdate();
int main()
{
    char command[10];
    init();
    calcDepthAndWeight();
    for (int i=1; i<=N; i++) buildJump(i);
    for (int i=1; i<=NumberOfTasks; i++)
    {
        scanf("%s", command);
        if (!strcmp(command, C_NAME)) solveQuery();
        else doUpdate();
    }
    return 0;
}

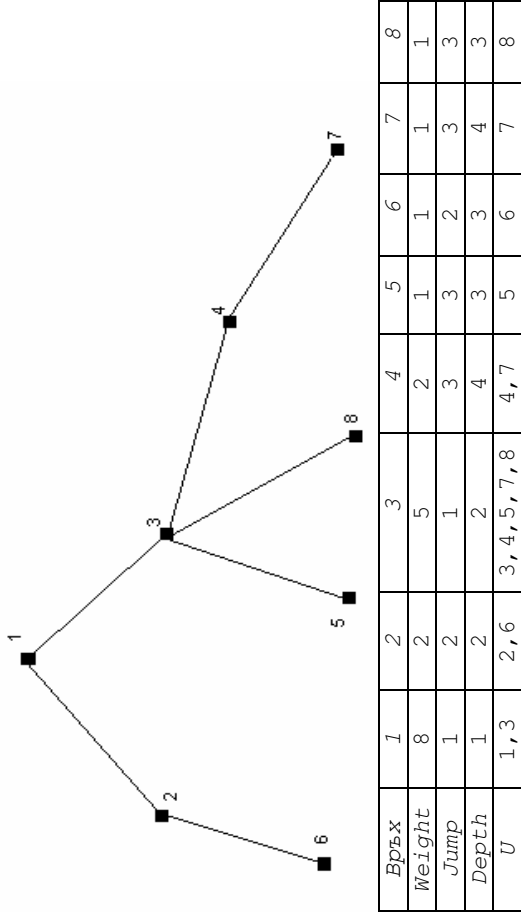
void init()
{
    scanf("%d%d", &N, &NumberOfTasks);
    for (int i=2; i<=N; i++)
    {
        int father;
        scanf("%d", &father);
        Root[i]=father; A[father].push_back(i);
    }
    stack<int> dfs;
```

```
За всеки връх  $A$ 
{ За всеки наследник  $B$  на  $A$ 
{
     $\text{Weight}(A) = \text{Weight}(A) + \text{Weight}(B)$ ;
     $\text{Depth}(B) = 1 + \text{Depth}(A)$ ;
}
}

/* С обхождане на дървото в дълбочина, тръгвайки от корена
пресмятаме стойностите  $\text{Jump}(A)$  за всеки връх  $A$  */
 $\text{Jump}(1) = 1$ ;
За всеки връх  $A \neq \text{корена}$ 
{
     $\text{Jump}(A) = \text{Jump}(\tau(A))$ ;
    Докато  $(\text{Weight}(\text{Jump}(A)) - \text{Weight}(A) > M)$ 
    {
         $\text{Jump}(A) = \text{наследника на } \text{Jump}(A) \text{ в}$ 
         $\pi(A, \text{Jump}(A))$ ;
    }
}
```

Сложността на този алгоритъм е $O(N)$, тъй като дърво с N върха има точно $N-1$ ребра. При решаването на задачата трябва да се обърне внимание, че при използване на рекурсия, системният стек може да се окаже недостатъчен. Затова авторовото решение реализира обхожданията на дървото в дълбочина, използвайки контейнера **stack** от STL.

Ако за примера в условието изберем $M = 5$, то тогава ще имаме:



Задача А2. РАЗСТОЯНИЕ

Класът на Елеонора е алски ШУМЕН (в сравнение с него, дори Големият Взрив е като слабо пукване). Главната причина за това е, че всеки в класа се опитва да говори със свой съученик, който, в повечето случаи, е в другата част на стаята. А както е широко известно – когато двама събеседници си говорят, шумът е пропорционален на квадрата от разстоянието между тях.

Един ден Ели се запита, каква е максималната олесия, която може да се постигне в нейния клас. За да се определи това, е достатъчно да се знаят разстоянията между всеки два събеседника. Мярка за олесията ще бъде сборът от квадратите на всички такива разстояния. Тъй като става дума за нормално българско училище – всички ученици си говорят по време на час. Класът на Ели се състои от **22** ученика.

Понякога някои ученици отсъстват, но **N**-те ученика в клас винаги са четен брой и всеки ученик води точно един разговор с някой от останалите **N-1**, като двамата си говорят помежду си и с никой друг.

Ели е самата прецизност, затова мери разстоянията в метри. Класната стая е квадрат със страна **10 000** метра.

Позицията на един ученик се задава с координатите му, като за начало на координатната система **(0, 0)** е избран един от ъглите на стаята (мястото на учителя). Задачата е да се напише програма **distance**, която по разположението на **N**-те ученика в класната стая, да определи максималния възможен шум, който може да се постигне.

Вход.

На първия ред на стандартния вход ще бъде зададен броят **N** на учениците ($2 \leq N \leq 22$, **N** – четно). На всеки от следващите **N** реда са зададени, разделени с един интервал, по две цели числа **x** и **y** ($0 \leq x, y \leq 10\,000$) – координатите на поредния ученик. Възможно е да има ученици с еднакви координати – например Пет'р (така изписано, според него, името му изглежда по-хакерско) много обича да седи на главата на Брус Веселия, защото Брус има лаптоп и го носи в клас.

Изход.

На единствения ред на стандартния изход програмата трябва да изведе максималната възможна олесия, с точност 3 знака след десетичната точка.

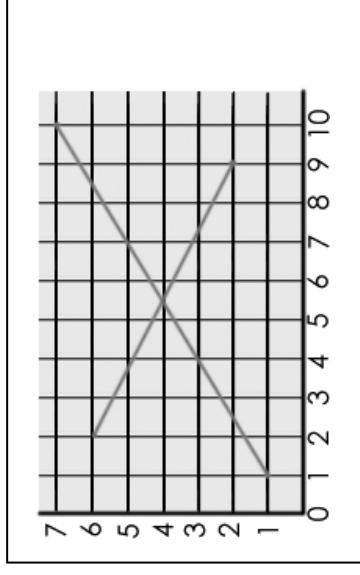
```
int state[MAXN];
void calcDepthAndWeight()
{
    Depth[1]=1; dfs.push(1);
    while (!dfs.empty())
    {
        const int v = dfs.top();
        if (!state[v]) Weight[v]=1;
        if (state[v] == (int)A[v].size())
        { dfs.pop(); Weight[Root[v]]+=Weight[v]; }
        else
        {
            const int to=A[v][state[v]];
            Depth[to]=Depth[v]+1;
            dfs.push(to); state[v]++; }
    }
}

void buildJump(const int v)
{
    int w=1, cr=Root[v], pred=v;
    do
    {
        U[pred].push_back(v);
        w+=Weight[cr]-Weight[pred];
        if (w>SQRT) break;
        pred=cr; cr=Root[cr];
    } while (pred);
    Jump[v]=cr;
}

void solveQuery()
{
    int time, from, to;
    scanf("%d%d", &time, &from, &to);
    int maxPumped=0, v=from;
    while (Depth[v] >= Depth[to])
    {
        if (Depth[Jump[v]] >= Depth[to] - 1)
        { maxPumped=max(maxPumped, F[v]); v=Jump[v]; }
        else { maxPumped=max(maxPumped, Water[v]); v=Root[v]; }
    }
    printf("%d\n", time*Q-maxPumped);
}

void doUpdate()
{
    int time, from, volume;
    scanf("%d%d", &time, &from, &volume);
    Water[from] += volume;
    for(int i=0; i<(int)U[from].size(); i++)
        F[U[from][i]]=max(F[U[from][i]], Water[from]);
}
```

автор: Ангел Джигаров



Пример:

Вход	Изход	Обяснение:
4	182.000	При данните от примерния вход най-голям сбор се получава, когато ученикът в точка (1,1) си говори с този в точка (10,7) – квадратът на разстоянието е 117.00, а този в точка (2,6) си говори с този в точка (9,2) – квадратът на разстоянието е 65.00. Тогава сборът е 117.00 + 65.00 = 182.00.

Решение:

Тъй като задачата е **NP**-пълна и няма известен алгоритъм, който да я решава за полиномиално време, то трябва да направим пълно изчерпване. Но ограниченията са твърде големи и затова неинтелигентно изчерпване би било много бавно. Ето защо ще трябва да създадем алгоритъм, базиран на схемата динамично оптимиране. Би могло да се напише решение по схемата **greedy**, но то не може да разчита на горлям брой точки.

Ако направим наблюдението, че след като изберем първите две точки (т.е. двама ученика, които ще говорят помежду си), остават четен брой точки (**N-2** ученика), за които търсим отново максималната възможна сума, т.е. получаваме същата задача, но върху по-малък брой ученици). Следователно трябва да опишем функция, която решава зададената задача и получава като параметър множеството от точки, измежду които може да избира. За да посочим кои от точките вече са използвани и кои не са, ще използваме т.н. „битова маска”. Това ще рече – в една цяла променлива за всяка от точките определяме по един бит, който да показва дали съответната точка е използвана. При стойност **1** на бита с номер **I** в променливата – точката с номер **I** е вече използвана, а при стойност на бита нула – точката не е използвана. Следователно, когато функцията получи като параметър битовата маска, може да използва само точките от тези позиции, в които битовете са нули.

Например, достигнали сме до някаква ситуация и битовата маска сочи числото **2835**. Неговото представяне в двоична бройна система е: **101100010011**. Следователно вече сме използвали точки на позиции **0,1,4,8,9** и **11** (номерацията на битовете започва от **0**, като битът с номер **0** е най-младшият бит в двоичното представяне).

Реално погледнато функцията не се нуждае от нищо друго, освен кои точки може да използва и кога да спре. Тъй като от броя на битове със стойност **1** може да се определи колко точки вече са обработени и колко остават, не е нужно на функцията да се подава и този брой, но за удобство (за да не броем всеки път единичните битове в маската) можем да го подаваме на функцията, като втори параметър. Да означим с **d(x,y)** квадрата на разстоянието между точките **x** и **y**. Динамичното оптимиране ще реализираме под формата на рекурсия със запомняне на резултатите от решените подзадачи в масива **T[]**. Рекурсивната ни функция ще изглежда така

```
recurse(mask, level)
{
    if (level>maxlevel) return 0;
    if (T[mask]!=0) return T[mask];
    for (every x,y not in set)
    {
        new_mask=unset(x,y,mask);best_value=0;
        f=d(x,y)+recurse(new_mask,level+1);
        if (f>best_value) best_value=f;
    }
    T[mask]=best_value;
    return best_value;
}
```

В задачата има 2 уловки. Първо в най-лошия случай може да имаме **11** двойки, които са на разстояние **10000*2+10000^2**, чийто общ сбор е **11*200 000 000=2 200 000 000** – т.е. стойност, която е малко над ограниченията на типа **int**. Има две решения на проблема – или да използваме **unsigned int** (тъй като нямаме отрицателни числа) или да използваме **long long**. Второто решение обаче не решава проблема, защото както ще видим по-долу, не можем да спазим ограничениято за използваната памет.

Така стигаме и до втория проблем. Само за да пазим таблицата с вече изчислени стойности, които са решенията на подзадачите, ще са ни нужни **2²²=4 194 304** полета от паметта. Ако използваме **long long**, то само за масива **T[]** ще са ни нужни **32 MB** памет (при ограничение **16 MB**). Нещо повече – дори при решението да използваме **unsigned int** ще са ни нужни точно **16** мегабайта (а ще ни трябват и допълнителни променливи, масив, където да пазим точките и т.н.). И така – паметта не достига.

При внимателно обмисляне можем да забележим, че тъй като задължително имаме четен брой битове в маската (използваните точки са комбинирани по двойки), то от младшите **21** бита можем еднозначно да определим **22**-рия. Ако в младшите **21** бита имаме четен брой единици – значи последния е нула; в противен случай е **1**. Това ни дава възможност да направим таблицата за запомняне на резултатите два пъти по-малка и така да намалим необходимата памет с **8 MB**. Сложността на описания алгоритъм е **O(N² * 2^N)**.

Нецелочисления отговор е даден само за да се провери внимателно ли са прочели условиято състезателите.

автор: *Александър Андриев*

Задача А3. ГОЛЯМО ЧИСЛО

Посвещава се на 300-годишнината от рождението на великия математик Леонард Ойлер.

Напишете програма **bignum**, която намира най-голямото число в q -ична бройна система, такова, че всичките k -цифрени числа, образувани от k последователни цифри на числото да бъдат различни.

Вход.

От стандартния вход се въвеждат q и k .

Изход.

Търсеното число да се изведе на стандартния изход.

Ограничения:

$2 \leq q \leq 10$, $1 \leq k \leq 14$, броят на цифрите в търсеното число е по-малък от 20000.

ПРИМЕР 1

Вход
2 3

Изход
1110100011

ПРИМЕР 2

Вход
3 1

Изход
210

Решение:

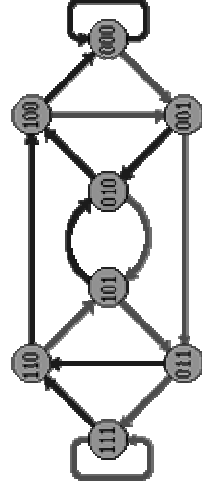
Различните q -ични низове с дължина k са q^k на брой. Низ с дължина n има $n-k+1$ подниза с дължина k . Следователно, за търсеното решение е в сила $n-k+1 \leq q^k$, откъдето $n \leq q^k + k - 1$.

Разглеждаме ориентиран граф с върхове всички q -ични низове с дължина k – 1. От връх **A** има ребро към връх **B**, когато последните $k-2$ цифри на **A** съвпадат с първите $k-2$ цифри на **B**.

Удобно е да си мислим, че реброто от **A** към **B** е надписано с последната цифра **B**.

Така от всеки връх в графа излизат по q ребра, надписани с цифрите $0, 1, \dots, q-1$. Във всеки връх влизат също по q ребра.

Например, при $q=2$ и $k=4$, графът има следния вид:



В ориентиран граф, в който за всеки връх броят на влизащите ребра е равен на броя на излизащите ребра има ойлеров цикъл.

Всеки ойлеров цикъл в разглеждания граф определя низ, който съдържа точно по веднъж всеки от q -ичните низове с дължина k . Записваме низа от началния връх и след това етикетите на ребрата, през които минава ойлеровия цикъл. Полученият низ е с дължина $n = q^k + k - 1$.

Например от ойлеровия цикъл

111-111-110-100-000-001-010-100-001-011-110-101-010-101-011-111
получаваме 111-1-0-0-0-0-1-0-0-1-0-1-0-1-1-1 или 111000010011010111.

Обратно, низ, съдържащ еднократно всички q -ични низове с дължина k , определя ойлеров цикъл в графа.

Следователно задачата е еквивалентна с намирането на ойлеровия цикъл в графа, който определя лексикографски най-големия низ.

Прилагаме стандартния алгоритъм за намиране на ойлеров цикъл, основан на търсене с връщане.

Започваме от върха съдържащ най-голямото число. Когато от някой връх има няколко ребра, по които може да продължим, избираме реброто, надписано с най-голяма цифра.

За конкретния пример получаваме:

111-111-110-101-011-111 – няма продължение, връщаме се назад до връх, от който излизат ребра

111-111-110-101-011-111

111-111-110-101-011-110-100-001-011 – връщане назад - -111

111-111-110-101-011-110-100-001-011-111

111-111-110-101-011-110-100-001-010-101-010-100-000-001-011-111

111-111-110-101-011-110-100-001-010-101-010-100-000-001-011-111

111-111-110-101-011-110-100-001-010-101-010-100-000-000-001-011-111

Търсеното число е

111-1-0-1-1-0-0-1-0-1-0-0-0-0-1-1-1 или 111011001010000111.

Броят на върховете в графа е $m = q^k - 1$.

Броят на ребрата в графа е $m \cdot q = q^k$.

Алгоритъмът за търсене на ойлеров цикъл има линейна сложност по броя на ребрата.

Така получаваме времева сложност $O(q^k)$.

При това, по условие $n = q^k + k - 1 < 20000$, следователно $q^k < 20000$.

За представяне на графа използваме масив **a**, в който елементът **a[x]** представлява броя на необходимите ребра, излизащи от връх **x**.

автор: Стоян Капралов

Задача В1. ПЕРМУТАЦИИ

Пермутация наричаме функция $f: \{1, 2, \dots, n\} \rightarrow \{1, 2, \dots, n\}$, за която $f(i) \neq f(j)$ при $i \neq j$.

На всеки две пермутации f и g съпоставяме пермутация h , такава, че $h(i) = f(g(i))$, за $i=1, \dots, n$. Пермутацията h наричаме *произведение на f и g* и записваме $h = fg$.

Нека $f(1) = i_2, f(i_2) = i_3, \dots, f(i_{n-1}) = i_n$. Ако $i_k \neq 1$, за $k=2, 3, \dots, n$, казваме, че пермутацията f е *цикъл с дължина n* и записваме $f = (1, i_2, i_3, \dots, i_{n-1}, i_n)$.

Напишете програма **PERM**, която по дадена пермутация f , определя дали f е цикъл с дължина n или произведение на два цикъла с дължина n .

Вход

От първия ред на стандартния вход се въвежда n .

От втория ред на стандартния вход се въвеждат числата $f(1), f(2), \dots, f(n)$.

Изход

Ако пермутацията f не е цикъл с дължина n , нито произведение на два цикъла с дължина n , на стандартния изход да се изведе 0.

Ако f е цикъл с дължина n , т.е. $f = (1, i_2, i_3, \dots, i_{n-1}, i_n)$, на първия ред на стандартния изход да се изведе 1, а на втория ред – числата $1, i_2, i_3, \dots, i_{n-1}, i_n$, разделени с по един интервал.

Ако f не е цикъл с дължина n , но $f = gh$, където $g = (1, i_2, i_3, \dots, i_{n-1}, i_n)$ и $h = (1, j_2, j_3, \dots, j_{n-1}, j_n)$ са цикли с дължина n , на първия ред на стандартния изход да се изведе 2, на втория ред – числата $1, i_2, i_3, \dots, i_{n-1}, i_n$ и на третия ред – числата $1, j_2, j_3, \dots, j_{n-1}, j_n$.

Ограничения

$3 \leq n \leq 10000$ (поне в 30% от тестовете $n \leq 10$)

ПРИМЕРИ

ПРИМЕР 1	ПРИМЕР 2	ПРИМЕР 3
Вход	Вход	Вход
5	5	5
2 3 4 1 5	3 4 5 1 2	3 2 4 1 5
Изход	Изход	Изход
0	1 1 3 5 2 4	2 1 4 2 3 5 3 1 2 4 5

Решение:

Формално въведение

Пермутация (на n елемента), $n \geq 1$, наричаме всяко взаимнооднозначно преобразуване $f: \{1, 2, \dots, n\} \rightarrow \{1, 2, \dots, n\}$. Множеството от всички пермутации на n елемента (при фиксирано n) означаваме със S_n .

В S_n въвеждаме операция *композиция*: на всеки две пермутации f и g от S_n съпоставяме пермутацията $f \circ g$ от S_n , която се дефинира по следното правило: ($f \circ g$) (i) = $f(g(i))$, $i=1, \dots, n$. (Естествено е да четем $f \circ g$ като “еф след же”).

Пермутацията $\text{id} \in S_n$ дефинирана с $\text{id}(i) = i$, $i=1, \dots, n$, наричаме *идентитет*. Тук е удачно в означението да включим индексът n , тъй като идентитетите в S_n при различни n са различни, но ще считаме, че той се подразбира от контекста.

Свойства на операцията (които се проверяват тривиално):

1. Асоциативност – $(f \circ g) \circ h = f \circ (g \circ h)$ за всички f, g, h от S_n .
2. Идентитетът е неутрален елемент – $f \circ \text{id} = \text{id} \circ f = f$ за всяко f от S_n .
3. Съществуване на обратни – за всяко f от S_n съществува (единствено) f^{-1} от S_n , такава че $f \circ f^{-1} = f^{-1} \circ f = \text{id}$.

За по-нагледно представяне пермутацията f от S_n , такава че $f(i) = k_i$, $i=1, \dots, n$ записваме с таблица с два реда по следния начин:

$$\begin{pmatrix} 1 & 2 & \dots & n \\ k_1 & k_2 & \dots & k_n \end{pmatrix}.$$

Естествено, тъй като f е пермутация от S_n , на долния ред в таблицата стои просто някакво разместване на числата $1, 2, \dots, n$.

Когато пермутациите са представени по този начин, операциите върху тях се извършват лесно и ефективно.

Ще дадем примери (в S_5):

$$\text{id} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix}, \text{ нека } f = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 5 & 1 & 4 & 3 \end{pmatrix}, g = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 5 & 3 & 1 & 2 \end{pmatrix}.$$

Тогава $f \circ g = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 3 & 1 & 2 & 5 \end{pmatrix}$, по-подробно, прилагаме g , след това f – например,

4 отива чрез g в 1 и после чрез f в 2.

Също, $g \circ f = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 5 & 2 & 4 & 1 & 3 \end{pmatrix}$. От този пример се вижда, че $f \circ g \neq g \circ f$, т.е.

композицията в общия случай не е комутативна.

$$\text{Имаме } f^{-1} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 1 & 5 & 4 & 2 \end{pmatrix},$$

получаваме $\begin{pmatrix} 2 & 5 & 1 & 4 & 3 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix}$ след което сортираме стълбовете, така че да се получи представяне на пермутация.

Цикъл наричаме пермутация f в S_n , за която съществува $k \geq 1$ и различни числа $i_1, \dots, i_k \in \{1, 2, \dots, n\}$, такива че действието на f се описва така: $f(i_1) = i_2, f(i_2) = i_3, \dots, f(i_{k-1}) = i_k, f(i_k) = i_1, f(i) = i$ за $i \neq i_j$. Числото k наричаме дължина на цикъла.

Цикълът f означаваме по следния начин: $(i_1 i_2 \dots i_k)$. Отново считаме, че индексът n се подразбира. Също, този запис не е еднозначен, например $(i_1 i_2 \dots i_k) = (i_2 i_3 \dots i_k i_1)$. По-общо, всеки цикъл с дължина $k \geq 2$ може да се запише по точно k различни начина.

Например, пермутацията $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 5 & 1 & 4 & 3 \end{pmatrix}$ от S_5 е цикълът (1253) , който има дължина 4 и той може да се запише още по следните начини: (2531) , (5312) и (3125) .

Тривиално, пермутацията id от S_n е цикъл с дължина 1 и може да се запише по n различни начина, $id = (1) = (2) = \dots = (n)$.

Ако $(i_1 \dots i_k)$ е цикъл с дължина k , тривиално се съобразява, че $(i_1 \dots i_k)^{-1}$ също е цикъл с дължина k , а именно $(i_1 \dots i_k)^{-1} = (i_k \dots i_1)$.

Два цикъла $(i_1 \dots i_k)$ и $(j_1 \dots j_\ell)$ наричаме *независими*, точно когато $\{i_1, \dots, i_k\} \cap \{j_1, \dots, j_\ell\} = \emptyset$. Лесно се съобразява, че два независими цикъла винаги комутират, т.е. $(i_1 \dots i_k) \circ (j_1 \dots j_\ell) = (j_1 \dots j_\ell) \circ (i_1 \dots i_k)$.

Например (в S_5): $(123) \circ (45) = (45) \circ (123) = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 1 & 5 & 4 \end{pmatrix}$.

Теорема: Всяка пермутация в S_n различна от id се представя като композиция на независими цикли с дължина поне 2. Представянето е единствено, с точност до подредба на множителите.

Няма да доказваме теоремата, а ще дадем пример как се получава представянето. Да вземем следната пермутация от S_{11} :

$$f = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 4 & 7 & 3 & 9 & 1 & 10 & 5 & 6 & 11 & 8 & 2 \end{pmatrix}.$$

Започваме от 1 и прилагаме към него последователно f : $1 \rightarrow 4 \rightarrow 9 \rightarrow 11 \rightarrow 2 \rightarrow 7 \rightarrow 5 \rightarrow 1$ – със сигурност след краен брой стъпки ще достигнем точно до първия елемент (1), от който сме започнали. Тогава първият множител в представянето е цикълът $(1\ 4\ 9\ 11\ 2\ 7\ 5)$. След това взимаме първият неизползван символ (в случая 3) и извършваме същото, т.е. започваме от 3 и прилагаме към него последователно f : $3 \rightarrow 3$ – тук спираме още на първата стъпка и получаваме цикъл с дължина 1, който обаче не се включва в представянето.

По-нататък взимаме 6 : $6 \rightarrow 10 \rightarrow 8 \rightarrow 6$ и добавяме множител $(6\ 10\ 8)$ в представянето.

Вече преминахме през всички символи, така че разполагаме с отговорът: $f = (1\ 4\ 9\ 11\ 2\ 7\ 5) \circ (6\ 10\ 8)$.

Транспозиция наричаме пермутация (от S_n), която е цикъл с дължина 2, т.е. която се представя като (ij) , където i и j са различни числа от $\{1, 2, \dots, n\}$. Ясно е, че за всяка транспозиция (ij) са изпълнени $(ij) = (ji)$, $(ij) \circ (ij) = id$, $(ij)^{-1} = (ij)$.

Твърдение: Всяка пермутация от S_n се представя като композиция на транспозиции. Във всяко такова представяне четността на броя на транспозициите е една и съща.

Отново няма да доказваме твърдението. Идеята е да разложим пермутацията като композиция на независими цикли и после да разложим всеки цикъл на транспозиции като използваме следното равенство: $(i_1 i_2 \dots i_k) = (i_1 i_k) \circ (i_1 i_{k-1}) \circ \dots \circ (i_1 i_2)$. Представянето на пермутация като композиция на транспозиции очевидно не е единствено (например, $f = f \circ (12) \circ (12)$ за всяка f от S_n при $n \geq 2$ или по-нетривиално, в S_4 , $(1234) = (14) \circ (13) \circ (12) = (12) \circ (23) \circ (34)$). По-нетривиалната част от твърдението е да се провери, че четността на броя на транспозициите във всяко представяне е една и съща. Това може да се направи по няколко начина, най-директно, като се разгледа действието на S_n върху подходящ антисиметричен полином на n променливи (т.е. полином, който променя знака си при смяна на местата на две от променливите). Но ние ще използваме наготово този факт.

Една пермутация f от S_n е *четна* (*нечетна*) точно тогава, когато във всяко представяне на f като композиция от транспозиции, техният брой е четен (нечетен). Например, един цикъл с дължина k е четна (нечетна) пермутация точно тогава, когато k е нечетно (четно) число. Така транспозициите са нечетни пермутации.

Тривиално се съобразява, че ако f и g (от S_n) са четни пермутации, то $f \circ g$ и f^{-1} също са четни. (f^{-1} е четна, тъй като обратната пермутация на транспозиция е транспозиция и е изпълнено равенството: $(f_1 \circ f_2 \circ \dots \circ f_k)^{-1} = f_k^{-1} \circ f_{k-1}^{-1} \circ \dots \circ f_1^{-1}$.)

Когато знаем представянето на една пермутация като композиция на независими цикли лесно и ефективно можем да определим нейната четност – а именно, определяме четността на всеки един от циклите (директно от тяхната дължина) и след това използваме следните (тривиални) правила:

четна $\circ$ *четна* = *четна*,

четна $\circ$ *нечетна* = *нечетна*,

нечетна $\circ$ *четна* = *нечетна*,

нечетна $\circ$ *нечетна* = *четна*.

Всичко, което изложихме дотук имаше за цел да станат възможни формулировката и доказателството на следващата

Теорема: Всяка четна пермутация от S_n се представя като композиция на точно два цикъла с дължина n .

Тук привеждаме доказателство, тъй като ще го използваме съществено по-долу.

Доказателство. Провеждаме индукция по n . Случаите $n=1$ и $n=2$ са тривиални.

Именно, при $n=1$ имаме единствената (четна) пермутация $(1) = (1) \circ (1)$, а при $n=2$ имаме точно две пермутации, от които точно една е четна – $(1) = (12) \circ (12)$.

Нека сега да предположим, че $n > 2$ и нека f е четна пермутация от S_n . Без ограничение на общостта можем да считаме, че $f(1) = n$. (Ако $f(1) = n$, то $f(2) = n$ и можем вместо 1 да вземем 2).

Нека $x = f(1)$ ($x \neq n$) и $g = (xn) \circ f \circ (1n)$.

Тогава $g(n) = n \rightarrow 1 \rightarrow x \rightarrow n$ и следователно можем да разгледаме g като пермутация от S_{n-1} (като пренебрегнем действието и върху n). Очевидно е, че g е четна пермутация, тъй като f е четна. От индукционното предположение, $g = \sigma_1 \circ \sigma_2$, където σ_1 и σ_2 са цикли с дължина $n-1$. Естествено, можем да разгледаме σ_1 и σ_2 като пермутации от S_n , които оставят n неподвижно. При това положение, $f = (xn) \circ g \circ (1n) = (xn) \circ \sigma_1 \circ \sigma_2 \circ (1n)$ и веднага се съобразява, че $(xn) \circ \sigma_1$ и $\sigma_2 \circ (1n)$ са цикли с дължина n .

Обратното твърдение е очевидно, т.е. произведение на два цикъла с дължина n със сигурност е четна пермутация. Така, една пермутация f от S_n се представя като произведение на два цикъла с дължина n точно тогава когато f е четна.

След това (доста обемно) формално въведение имаме достатъчно знания за да атакуваме задачата.

В нея пермутацията f трябва да се представи като композиция на най-много два цикъла с дължина n . Последната теорема, която доказахме + някои тривиални съображения дават изчерпателно решение на задачата, а именно:

- f се представя като композиция на точно един цикъл с дължина n , точно тогава когато f е цикъл с дължина n , което тривиално може да се провери с едно (линейно) преминаване през пермутацията;
- f се представя като композиция на точно два цикъла с дължина n , точно тогава когато f е четна пермутация, като от доказателството по-горе се вижда как ефективно да се намери едно такова представяне (по метода разделяй и владей с линейно спускане по пермутацията).
- Оттук задачата няма решение точно тогава, когато f е нечетна пермутация и f не е цикъл с дължина n .

Реализация

Пермутацията f , съвсем естествено, представяме с едномерен масив $\text{perm}[n+1]$, ($\text{perm}[i] = f(i)$, $i=1, \dots, n$). За удобство съхраняваме и обратната пермутация $\text{invperm}[n+1]$, която тривиално се изчислява по следната схема:

```
for i = 1 ... n
    invperm[perm[i]] = i.
```

Следващата стъпка е да проверим дали f е цикъл с дължина n . Това отново става тривиално, ако използваме допълнителен булев едномерен масив $\text{vis}[n+1]$. Ако f е цикъл с дължина n , то сме готови, извеждаме решението и приключваме.

В противен случай, стартираме рекурсивна процедура $\text{compute}(n)$.

Процедурата compute при $n > 2$ избира $s=1$ или $s=2$, така че $f(s) \neq n$, съхранява $x=f(s)$ и изчислява $(xn) \circ f \circ (sn)$, което се свежда до няколко присвоения (но за това съществено се използва явното представяне на обратната пермутация), по същия начин изчислява също

$$((xn) \circ f \circ (sn))^{-1} = (sn) \circ f^{-1} \circ (xn).$$

Следва рекурсивно извикване $\text{compute}(n-1)$. След връщане от рекурсията актуализираме два масива invansleft и ansright , в които се съхранява представянето на отговора, като invansleft съдържа обратния на първия цикъл и ansright съдържа самия втори цикъл. Именно, invansleft трябва да се композира отлясно с (xn) (тъй като използваме представяне на обратния цикъл) и ansright трябва да се композира отлясно със (sn) . Всяка от тези две композиции се свежда до три присвоения. Дъното на рекурсията е при $n=2$.

Но това пресмятането е тривиално – ако $f(1)=1$ и $f(2)=2$, то инициализираме invansleft и ansright да бъдат транспозицията (12) , а ако $f(1)=2$ и $f(2)=1$, то първоначалната пермутация е била нечетна и следователно извеждаме "No solution." и приключваме.

След приключване на изпълнението на всички рекурсивни извиквания, в invansleft се съдържа обратния на левия цикъл в отговора, така че трябва да го обрнем в ansleft (линейно) по начина, който вече коментирахме, а в ansright се съдържа десния цикъл в отговора. И така сме готови да изведем отговора.

автор: Иван Георгиев

Задача B2. ПРАВИ

Разглеждаме точките с целочислени координати (x, y) , за които $0 \leq x \leq a$ и $0 \leq y \leq b$. Напишете програма **LINES**, която намира колко прави минават през поне две от тези точки.

Вход

От един ред на стандартния вход се въвеждат a и b ($0 < a, b < 3000$).

Изход

На един ред на стандартния изход програмата трябва да изведе броя на търсените прави.

ПРИМЕР

Вход

2 2

Изход

20

Решение:

През дадените точки минават $b+1$ хоризонтални и $a+1$ вертикални прави. Остава да изброим всички останали прави. В следващите разглеждания ще се интересуваме само от прави, които минават през поне две от дадените точки, но не са хоризонтални или вертикални.

Решение 1

Една идея за намиране на този брой е следната: През всеки две точки минава една права. Има опасност обаче тази права да бъде преброена няколко пъти. За да не допуснем това, права през точките $A(x_1, y_1)$ и $B(x_2, y_2)$ броим само когато извън отсечката AB върху правата AB няма други от дадените точки. При това достатъчно е да намерим колко са правите AB , за които ако $x_1 < x_2$, то $y_1 < y_2$ (останалите са също толкова на брой).

Сложността на този алгоритъм е $O(n^2)$, където n е броят на всички точки.

```
#include <iostream>
using namespace std;
int nod(int a, int b)
{ int r = a%b;
  while (r > 0)
  { a = b; b = r; r = a%b; }
  return b;
}

int main()
{ int a, b;
  cin >> a >> b;

  long long cnt = 0;
  for(int x1=0; x1<a; x1++)
  for(int y1=0; y1<b; y1++)
    for(int x2=x1+1; x2<=a; x2++)
      for(int y2=y1+1; y2<=b; y2++)
      {
        int p = x2 - x1, q = y2 - y1;
        int d = nod(p,q);
        int dx = p/d, dy = q/d;
        int x0 = x1 - dx, y0 = y1 - dy;
        int x3 = x2 + dx, y3 = y2 + dy;
```

```
        if( (x0<0||y0<0) && (x3>a||y3>b) ) cnt++;
    }

    cnt = cnt * 2 + a + b + 2;

    cout << cnt << endl;

    return 0;
}
```

Решение 2

Нека m е по-голямото от числата a и b . Една права може да мине най-много през $m+1$ от разглежданите точки. Нека с $x[2], x[3], \dots, x[m+1]$ означим броя на правите, които минават съответно през $2, 3, \dots, m+1$ точки. Те могат да бъдат намерени по следния начин:

За всяка отсечка, която не е успоредна на координатните оси и краищата на която са две от дадените точки проверяваме колко от разглежданите точки лежат на нея. Ако краищата на тази отсечка са точките $A(x_1, y_1)$ и $B(x_2, y_2)$, броят на точките, които лежат на тази отсечка е равен на $\text{НОД}(|x_1 - x_2|, |y_1 - y_2|) + 1$. Интересуващите ни отсечки могат да бъдат разглеждани като диагоналите на всевъзможните правоъгълници със страни, успоредни на координатните оси и върхове – четири от дадените точки. При това броене обаче някои прави са броени по няколко пъти. Така например права, която минава през три точки е броена 2 пъти като права, минаваща през две точки; права, минаваща през четири точки е броена 2 пъти като права, минаваща през три точки и 3 пъти като права, минаваща през две точки и т.н. След извършване на съответните корекции получаваме истинските стойности на с $x[2], x[3], \dots, x[m+1]$.

Окончателно търсеният брой прави е равен на $x[2] + x[3] + \dots + x[m+1] + a + b + 2$.

Сложността на този алгоритъм е $O(n)$, където n е броят на всички точки.

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

const int mm=3100;
long long x[mm];

int nod(int a, int b)
{ if (a == 0) return b;
  if (b == 0) return a;
  if (a >= b) return nod(a-b,b);
  else return nod(a,b-a);
}
```

```

int main()
{ int a, b;
  cin >> a >> b;
  int m;
  m = a;
  if (b > m) m = b;
  for(int p=1; p<=a; p++)
    for(int q=1; q<=b; q++)
      { int s;
        s = nod(p,q);
        x[s+1] = x[s+1] + 2*(a-p+1)*(b-q+1);
      }
  long long sum = x[m+1];
  for(int i=m+1; i>=3; i--)
    { for(int j=i-1; j>=2; j--)
      x[j] = x[j] - (i-j+1)*x[i];
      sum = sum + x[i-1];
    }
  sum = sum+a+b+2;
  cout << sum << endl;
  return 0;
}

```

Решение 3

Нека правата **AB** има уравнение $y=kx+n$. Числото k се нарича ъглов коефициент на правата. Две прави са успоредни тогава и само тогава, когато техните ъглови коефициенти са равни. Ако $A(x_1, y_1)$ и $B(x_2, y_2)$, то $k = (y_2 - y_1) / (x_2 - x_1)$. Њгловият коефициент k е равен и на тангенса на ъгъла, който правата **AB** сключва с положителната абсцисна полуос.

В това решение ще изброим само правите, които имат положителен ъглов коефициент, защото правите с положителен и отрицателен ъглов коефициент, които минават през дадените точки са равен брой. Възможните положителни ъглови коефициенти в разглежданата задача може да определим като броя на всички несъкратими дроби от вида p/q , където p и q са положителни делители съответно на a и b . Остава да разберем колко прави имат съответния ъглов коефициент. За целта може да броим най-левите от дадените точки (с най-малката абсциса), които могат да лежат на права със съответния ъглов коефициент. Броят на тези точки е $(a-p+1) \cdot (b-q+1) - \max(a-2, p+1, 0) \cdot \max(b-2, q+1, 0)$.

Сложността на този алгоритъм отново е $O(n)$, където n е броят на всички точки, но програмата работи по-бързо от предложената в *Решение 2*.

```

#include <iostream>
using namespace std;

```

```

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

int main()
{ int a, b;
  cin >> a >> b;
  long long cnt = 0;

  for(int p=1; p <= a; p++)
  for(int q=1; q <= b; q++)
  if (nod(p,q)==1)
    cnt=cnt+ (a-p+1) * (b-q+1) -max(a-2*p+1,0) *max(b-2*q+1,0);
  cnt = cnt * 2 + a + b + 2;
  cout << cnt << endl;

  return 0;
}

```

Решение 4

Бързодействието на програмата може да се подобри като генерирането на ъгловите коефициенти на правите се ускори. Такава модификация на *Решение 3* е следващата програма. Сложността на алгоритъма отново е $O(n)$, където n е броят на всички точки.

```

#include <iostream>
using namespace std;

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

int main()
{ int a, b;
  cin >> a >> b;
  if(a>b) { int w = a; a = b; b = w; }
  long long cnt = 0;
  for(int p=1; p <= a; p++)
  for(int t=1; t <= p; t++)
    if(nod(p,t)==1)

```

```

for(int q = t; q<=b; q = q + p)
    cnt=cnt+(a-p+1)*(b-q+1)-max(a-2*p+1,0)*max(b-2*q+1,0);
cnt = cnt * 2 + a + b + 2;
cout << cnt << endl;

return 0;
}

```

автор: Младен Манев

Задача В3. СКОКОВЕ

В равнината са дадени n точки с целочислени координати.

Напишете програма **JUMPS**, която намира колко най-много различни точки може да посети един скакалец, като се движи по дадените точки, ако от точка $A(x_1, y_1)$ може да скочи в точка $B(x_2, y_2)$, само когато $x_1 \leq x_2$ и $y_1 \leq y_2$.

Вход

На първия ред на стандартния вход е записано числото n ($1 \leq n \leq 10^6$). На следващите n реда на стандартния вход за записани координатите на дадените точки – цели неотрицателни числа, по-малки от 10^9 .

Изход

На стандартния изход да се изведе броят на точките в най-дългия възможен път на скакалеца, в който няма повторение на точки.

ПРИМЕР

Вход	Изход	Обяснение
5	3	$(0, 0) - (5, 0) - (6, 1)$
3 2		
5 0		
0 0		
6 1		
5 0		

Решение:

Първи начин.

Разглеждаме ориентиран граф G , дефиниран по следния начин: върховете на графа са дадените точки; от връх A има ребро към връх B , когато точките са различни и скакалецът може да скочи от A в B . Така задачата се трансформира в търсене на най-дълъг път в ориентиран граф без цикли. Топологичното сортиране на върховете може да се реализира посредством сортиране на точките в лексикографски ред (по x -

координатите, а при равни x -координати – по y -координатите). След това последователно пресмятаме $d[i]$ – дължината на най-дългия път в графа (по брой точки), завършващ в i -тата точка: $d[i] = 1 + \max d[j]$, където $j < i$ и има ребро от j -тата към i -тата точка. Сложността на алгоритъма е $O(n^2)$.

Втори начин.

1) Компресиране y -координатите на точките, така че най-малката y -координата да е 1, следващата – 2 и т.н. Нека след компресирането най-голямата y -координата да е m .

2) Сортираме точките в лексикографски ред (по x -координатите, а при равни x -координати – по y -координатите). Сега една точка $A_j(x_j, y_j)$ може да бъдат предшественик на точката $A_i(x_i, y_i)$ по най-дългия път, завършващ в A_i , когато $j < i$ и $y_j \leq y_i$.

3) Реализираме структура с две операции:

а) **update** (y, d) – отразява в структурата, че е намерен път с дължина d до точка A с y -координата равна на y ;

б) **getMax** (y) връща дължината на най-дългия, намерен досега, път в G , завършващ в някоя точка с y -координата по-малка или равна на y .

4) Тогава вечно се решава с изпълнението на следния програмен фрагмент:

```

update(A[0].y, 1);
for(int i=1; i<n; i++)
    if(A[i-1].x < A[i].x || A[i-1].y < A[i].y)
        update(A[i].y, 1+getMax(A[i].y));
ans = getMax(m);

```

За реализиране на структурата използваме индексно дърво с m елемента. Благодарение на компресирането на координатите имаме $m \leq n \leq 10^6$. Всяка операция с индексното дърво се изпълнява за време $O(\log n)$, а стъпка 4 за време $O(n \log n)$.

Всяка от стъпки 1, 2 и 4 се изпълнява за време $O(n \log n)$. Следователно и сложността на целия алгоритъм е $O(n \log n)$.

автор: Стоян Капралов

Задача С1. ЧИСЛОВА ТАБЛИЦА

Във всяка от клетките на правоъгълна таблица с m реда и n стълба е записано по едно положително цяло число, което е по-малко от 100. Три клетки на таблицата ще наричаме **съседни**, ако всяка от тях има обща страна с поне една от останалите две.

Да се напише програма **MAX1**, която намира най-голямата сума, която може да се получи като се съберат числа, записани в три **съседни** клетки на таблицата.

Вход

От един ред на стандартния вход се въвеждат **m** и **n** ($1 < m, n < 10$). От следващите **m** реда на стандартния вход се въвеждат по **n** числа, представляващи поредния ред от таблицата.

Изход

На един ред на стандартния изход програмата трябва да изведе търсената най-голяма сума.

ПРИМЕР

Вход

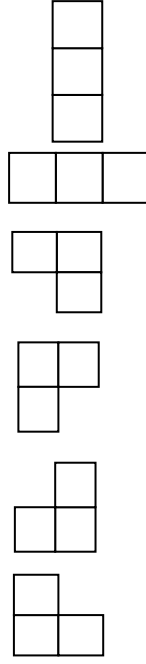
```
2 2
1 3
2 4
```

Изход

```
9
```

Решение:

От дефиницията за съседни клетки следва, че съществуват следните шест вида съседни клетки:



Числата в таблицата може да разглеждаме като елементи на двумерен масив. Обхождайки елементите на масива, пресмятаме всяка от интересувашите ни суми и определяме коя е най-голямата. За опростяване на разглежданията може да считаме, че в таблицата има още два допълнителни реда и два допълнителни стълба, елементите на които са **0**. Лесно се забелязва, че ако в някоя от образуваните суми участва нулев елемент от допълнителните редове или стълбове, то получената сума няма да бъде най-голямата.

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

```
int a[20][20];
```

```
int main()
{
    int maxi=0;
    int m, n;
    cin >> m >> n;
    for(int i=1; i<=m; i++)
        for(int j=1; j<=n; j++)
            cin >> a[i][j];
    for(int i=1; i<=m; i++)
        for(int j=1; j<=n; j++)
            {
                int s1, s2, s3, s4, s5, s6;
                s1 = a[i][j] + a[i][j+1] + a[i+1][j];
                s2 = a[i][j] + a[i][j+1] + a[i+1][j+1];
                s3 = a[i][j] + a[i+1][j] + a[i+1][j+1];
                s4 = a[i][j+1] + a[i+1][j] + a[i+1][j+1];
                s5 = a[i][j] + a[i][j+1] + a[i][j+2];
                s6 = a[i][j] + a[i+1][j] + a[i+2][j];
                if (s1 > maxi) maxi = s1;
                if (s2 > maxi) maxi = s2;
                if (s3 > maxi) maxi = s3;
                if (s4 > maxi) maxi = s4;
                if (s5 > maxi) maxi = s5;
                if (s6 > maxi) maxi = s6;
            }
    cout << maxi << endl;
    return 0;
}
```

автор: **Младен Манев**

Задача C2. РЕБУС

Вид числени ребуси са изрази съставени от букви и аритметични операции. Например такъв ребус е последователността

AB+B=BA

При тези ребуси на различни букви съпоставяме различни цифри, а на еднаквите букви – еднакви цифри. Решението е такава замяна на буквите с цифри, при която получаваме вярно числово равенство. За примера решение е **89+9=98**.

Напишете програма с име **REBUS**, решаваща числен ребус.

Вход

От единствения ред на стандартния вход се въвежда последователност от главни латински букви **'A', 'B', 'C'** и символите **'+' и '='**, представляваща числения ребус.

Изход

На един ред на стандартния изход програмата трябва да изведе едно решение на ребуса.

Ограничения

1. Последователността съдържа само главни латински букви 'A', 'B', 'C' и символите '+' и '=';
2. Броят символи в последователност, определяща число, е не повече от 9 символа;
3. В последователността има точно един символ '+';
4. В последователността има точно един символ '=' и той е след символа '+' в последователността на ребуса;
5. Ребуса винаги има решение;
6. Ако решенията са няколко, изведете кое да е;
7. В решението никое число не започва с "излишна" нула.

ПРИМЕР

Вход
AB+B=BA

Изход
89+9=98

Решение

Предложеното решение е с пълно изчерпване на всички възможности за всяка от буквите 'A', 'B', 'C' и прекъсване на изпълнението при намиране на първото вярно решение.

За целта последователността на ребуса се чете в низова променлива.

```
string eq;  
cin>>eq;
```

Генерирането на всички възможни стойности на буквите може да стане с

```
for (int a = 0; a < 10; a++)  
    for (int b = 0; b < 10; b++)  
        for (int c = 0; c < 10; c++)
```

Ако някои две цифри съвпадат имаме некоректно съпоставяне на букви и цифри и трябва да се премине към следващата цифра за съответната буква.

```
if (a==b || a==c || b==c) continue;
```

В противен случай – на всяка буква е съпоставена различна цифра и може да се премине към проверка за вярно решение с тези цифри. В променливите **Num1**, **Num2** и **Num3** ще създадем съответно първото, второто и трето число, а променливата **br** ще използваме, за да знаем кое число в момента генерираме. Не трябва да се забравя да се инициализират с нули при всеки нов опит.

Обхожда се целия низ и се отича, при достигане до '+' или '=' , че започва ново число.

```
int Num1 = 0, Num2 = 0, Num3 = 0, br = 0;  
for (int i = 0; i < eq.length(); i++)  
{  
    if (eq[i] == '+' || eq[i] == '=')  
    {  
        br++;  
        continue;  
    }  
}
```

В променливата **p** ще се съхрани стойността на цифрата, която съответства на поредната буква.

```
int p = 0;  
if (eq[i] == 'A') p = a;  
if (eq[i] == 'B') p = b;  
if (eq[i] == 'C') p = c;
```

Според брояча на обработените вече числа **br** се напругва цифрата **p** в съответното число (**Num1**, **Num2** или **Num3**)

```
if (br == 0) Num1 = Num1*10 + p;  
if (br == 1) Num2 = Num2*10 + p;  
if (br == 2) Num3 = Num3*10 + p;  
}
```

След като е обходен целият израз и са генерирани трите числа **Num1**, **Num2** и **Num3** се проверява дали има коректно решение. За целта трябва не само сумата на първите две да е равна на третото, но и да имаме правилният брой символи в заменения с цифри ребус. Функцията **cnt** преброява цифрите на число.

```
if (Num1 + Num2 == Num3)  
if (cnt(Num1) + cnt(Num2) + cnt(Num3) + 2 == eq.length())  
{  
    cout<<Num1<<'+ '<<Num2<< '=' <<Num3<<endl;  
    return 0;  
}
```

Примерна реализация

```
#include <iostream>  
#include <string>  
  
using namespace std;  
int cnt (int a);  
int main()  
{  
    string eq;  
    cin>>eq;  
    for (int a = 0; a < 10; a++)  
        for (int b = 0; b < 10; b++)
```

Задача С3. ИМЕНА

Елеонора е възмутена от сериалите, които баба ѝ гледа. За тридесет минути, колкото продължава само една от сериите, се споменават стотици действащи лица. Брук, която е бременна от Рич (той все още не знае), тайно се вижда с Ерик, но Стефани разбира и казва на Сали и Тейлър... Понякога на Ели имената до такава степен ѝ идват в повече, че тя не може да установи кой всъщност е главният герой.

Добре, че знае една особеност на сериалите – главният герой се споменава по-често от всички останали, взети заедно. За съжаление тя няма паметта на баба си, за да запомни всичките стотици хиляди (както ще се убедите от ограниченията по-долу, това е самата истина) герои, затова решила да си записва на листче всяко име, което чуе.

Тъй като Ели ви е приятелка (а освен това е готина :-), сте решили да ѝ помогнете да намери прословутия главен герой. Ели ви дава листчето (добре де, камарата от листи) с ***N***-те имена ($1 \leq N \leq 1\,000\,000$), всяко от което е не по-дълго от 20 знака, и... вие се оказвате в трудна ситуация. Очевидно няма да може да се справите без компютър и програмиране, тъй като вашата собствена памет е ограничена (**само IMB**). Не се предавайте! Напишете програма **NAMES**, която намира търсеното име.

Вход.

На първия ред на стандартния вход ще бъде зададено числото **N**, показващо колко имена е записала Ели. На всеки от следващите **N** реда ще има по едно име (не по-късо от 1 и не по-дълго от 20 знака), съставено от главни и малки латински букви. Гарантирано е, че ще има име, което да се среща повече пъти от всички останали имена, взети заедно. След последното име ще има знак за нов ред.

Изход.

На стандартния изход програмата трябва да изведе името на главното действащо лице на сериала.

ПРИМЕР

Примерен вход	Изход за примерния вход
11 Rich Rich Bruck Rich Eric Stephanie Rich Rich Sally Tailor Rich	Rich

```
for (int c = 0; c < 10; c++)
{
    if (a == b || a == c || b == c) continue;
    int Num1 = 0, Num2 = 0, Num3 = 0, br = 0;
    for (int i = 0; i < eq.length(); i++)
    {
        if (eq[i] == '+' || eq[i] == '-')
        {
            br++;
            continue;
        }
        int p = 0;
        if (eq[i] == 'A') p = a;
        if (eq[i] == 'B') p = b;
        if (eq[i] == 'C') p = c;
        if (br == 0) Num1 = Num1*10 + p;
        if (br == 1) Num2 = Num2*10 + p;
        if (br == 2) Num3 = Num3*10 + p;
    }
    if (Num1+Num2 == Num3)
        if (cnt (Num1)+cnt (Num2)+cnt (Num3)+2
            ==eq.length())
            cout<<Num1<<'+'+<<Num2<<'+'+<<Num3<<endl;
            return 0;
        }
    }
    cout <<"No answer.\n";
    return 0;
}

int cnt (int a)
{
    int br=0;
    do {
        br++;
        a/=10;
    } while (a);
    return br;
}
```

автор: Венета Богданова

Решение:

Задачата може да бъде решена по много различни начини, ако разполагаме с достатъчно памет. Елементарното решение е да се въведат всички имена в масив, да се сортират (независимо в какъв ред) и като резултат да се изведе средния елемент на масива (ако индексират е от 0 – този с индекс е $N/2$). При зададените ограничения (1 000 000 имена с максимална дължина 20) такова решение ще донесе не повече от 50% от точките.

Затова да подходим по друг начин – да записваме в стек прочетените имена и да реализираме следната операция с всеки прочетен от входа низ. Ако стекът е празен – добавяме безусловно прочетения низ в него. Ако новото име, прочетено от входа, съвпада с името което е на върха на стека – добавяме съпадащото име в стека. В противен случай, пропускаме прочетеното име и изваждаме от стека това, което е на върха му.

Твърдение. Името, което ще остане на върха на стека след прочитане на целия вход е търсеното най-често срещано име (мажорант или доминиращ низ).

Обяснението защо такъв алгоритъм работи коректно, се гради върху факта, че доминиращият низ (мажорантът) може да бъде изваден от стека, само “в двойка” с различен от него низ. Обратно – всеки от останалите низове ще бъде изваден от стека както от мажоранта, така и от произволен друг низ, различен от него. Затова, след прочитането на последното име и изпълнение на операцията, в стека не може да има низ, различен от мажоранта. Колкото до мажоранта, поне един екземпляр от него ще остане в стека, защото няма достатъчно други низове, които до извадят от стека всички копия на мажоранта.

Предложеното решение не е безпроблемно: ако в даден момент от входа постъпват твърде много еднакви имена, размерът на стека може да надхвърли ограниченията за памет. Забележете обаче, че в стека винаги имаме само копия на един и същ низ. Затова проблемът можем да решим по следния начин – вместо стек можем да запоним само низа, от който има копия в стека и броят на копия. При прочитане на поредния низ, в зависимост от това, дали е еднакъв или различен със запомнения – ще увеличаваме или намаляваме брояч. Когато при прочитане на поредната дума броячът е нула – тогава просто заменяме стария низ с новия и увеличаваме брояча на 1.

```
#include <stdio>
#include <cstring>
#define MAX 32

using namespace std;
int main(void)
{
    int i, n;
    char name[MAX]; int cnt;
```

```
char surname[MAX];

memset(name, 0, sizeof(name)); cnt = 0;
scanf("%d", &n); gets(surname);
for (i=0; i<n; i++)
{
    gets(surname);
    if (cnt == 0)
    {
        memcpy(name, surname, MAX); cnt = 1; continue;
    }
    if (!strcmp(surname, name)) cnt++; else cnt--;

    printf("%s", name);
    return 0;
}
```

автор: Александър Андиев

Задача D1. ЩАСТЛИВИ ГОДИНИ

Щастлива година е тази, която има в записа си точно две равни цифри. Годините са представени като цели положителни, четирисифрени числа. Напишете програма **HAPPY**, която отпечатва на екрана тази от годините, която не е щастлива и разликата между нея и някоя щастлива година е минимална.

Вход:

На първия ред от стандартния вход се прочита цяло число **N (0<N<9000)** – броя на годините. От следващия ред се прочитат **N** числа – самите години.

Изход:

На стандартния изход се отпечатва годината, която не е щастлива и е с минимална разлика с една от годините, които са щастливи. Гарантирано е, че има само една година, която не е щастлива и разликата и с някоя от щастливите години е минимална. Ако няма щастлива година или няма нещастлива година програмата трябва да отпечата 0.

Примери

Вход	Изход
4 1254 1987 1271 2002	1254
6 1234 2134 5678 9871 2341 2390	0

4	5103 5105 5104 5205				5104
---	---------------------	--	--	--	------

Забележка:

Търсената не щастлива дата може да е както по-голяма от някоя щастлива, така и по-малка.

Решение:

1 начин:

За всяка една от годините трябва да определим дали е щастлива или не. Нека отделим щастливите години в един масив, а нещастливите в друг. За всяка една година от масива на нещастливите години намираме разликата и по абсолютна стойност с всяка една година от масива на щастливите. Ако разликата е по-малка от минималната разлика намерена до момента запазваме годината в допълнителна променлива.

Този алгоритъм е от порядъка на $O(n^2)$;

II начин:

Както при първото решение така и тук трябва да определим коя година е щастлива и коя не е. Очевидно е, че ако сортираме годините по големина с едно обхождане на масива лесно можем да намерим търсената година. Използваме това, че годините са четирцифрени числа и реализираме сортиране чрез броеене. За целта ще ни е необходим масив, който е по-голям от **9999** елемента и в началото е нулиран. Нека в елемента на масива с индекс щастлива година присвоим стойност **1**, а на елемента на масива с индекс нещастлива година присвоим **2**. Използваме две променливи, в които да пазим последната щастлива години и последната нещастлива година до момента на обхождането. Обхождаме масива и ако стойността на поредния елемент е равна на **1**, намираме разликата на текущата година и последната нещастлива година, а в противен случай разликата на текущата и последната щастлива година. Ако тази разлика е по-малка от намерената до момента запазваме нещастливата година. Намираме разликата само в случай, че до момента на обхождането сме преминали през най-малко една щастлива и най-малко една нещастлива година.

Този алгоритъм е от порядъка на $O(n)$.

автор: Петър Петров

Задача D2. ИГРА НА ДУМИ

Напишете програма **WORDS**, която чете последователност от думи и съставя възможно най-дългата редица от думи, за която всяка следваща дума започва с

последната буква на предходната (**new** – **word** – **door** ...), като ако съществува такава дума, то тя е единствена . Всяка дума може да се използва само по веднъж.

Вход

От първия ред на стандартния вход се въвежда **n** – брой думи. На всеки от следващите **n** реда се въвежда по една дума.

Изход

На първия ред на стандартния изход се извежда число **b** - броя на думите в редицата и на всеки от следващите редове по една дума от нея.

Ако съществуват две редици с еднаква дължина се извежда тази, чиято първа дума е по-напред по азбучен ред.

Ако липсват думи, отговарящи на условието се извежда **0** .

Ограничения

$0 < n \leq 26$

максимална дължина на дума – **10** букви

ПРИМЕР

Вход

5

bear

new

door

sky

word

Изход

3

new

word

door

Решение:

Използват се два масива. Първият масив **words[26][10]** съдържа думите, като след прочитането всяка дума се записва в масива с индекс, отговарящ на поредността на първата ѝ буква в азбуката. В масива **used** се отбелязват вече използваните за получаването на редицата думи.

Прави се опит за генериране на редици от думи, започващи с всяка една от въведените думи. Редът **p=words[p][strlen(words[p])-1]-'a'**; изчислява индекса на епентуалната следваща дума от редицата, отговарящ на поредността на последната буква от текущата дума в азбуката. Ако такава дума съществува, тя се добавя в редицата и думата се отбелязва като използвана, в противен случай се прави опит за генериране на редица, започваща със следващата дума от масива **words**.

Дължината на всяка новополучена редица се сравнява с текущата най-дълга и ако е по-голяма се запомня индекс на началната дума, от която е била формирана.

автор: Веселина Спасова

Задача D3. ДЕЛИМОСТ

Разглеждаме всички цели числа, намиращи се в естествения ред на числата между две дадени цели положителни числа **a** и **b** (включително **a** и **b**). Напишете програма **DIV**, която определя колко от разглежданите числа се делят на дадено цяло число **c**.

Вход:

Програмата въвежда от стандартния вход стойностите на **a**, **b** и **c**, разделени с по един интервал.

Изход:

Програмата извежда на стандартния изход търсения брой като едно цяло число.

Ограничения:

0 < a < b < 1 000 000 000 000 000
1 < c < 1 000 000 000 000 000

ПРИМЕР

Вход

7 60 15

Изход

4

Решение:

При допустимо време за работа на програмата, всяка реализация на изчерваща проверка би решила задачата за брой на разглежданите числа, който не надминава няколко десетки милиона. За да решим задачата в общите рамки на ограниченията от условието ѝ, забелязваме, че числата между **a** и **b**, които се делят на **c** се срещат през равни интервали, всеки съдържащ по **c** на брой последователни числа. Търсеният брой се получава след преброяване на тези интервали. Това се пресмята от резултата на делението на броя на числата между **a** и **b** с делителя **c**. Подробености за програмната реализация читателят може да научи от текста на програмата:

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

long long int a,b,c;

int main()
{
    cin >> a >> b >> c;
    long long int al=a;
    if(a%c!=0) al=a+c-a%c;
    long long int bl=b-b%c;
    long long int r=0;
    if(al<=bl) r = 1+(bl-al)/c;
    cout << r << endl;
}
```

автор: Емил Келеведжиев

Задача E1. ЛИНИЙКА

Върху линейка с дължина **100000** см са нанесени деления през **1** см. Деленията на линейката са номерирани с поредни числа, като в началото и е записано числото **0**. Талантливо поставил върху линейката три летвички, които изработил в часовете по труд и техника. Всяка летвичка поставял на произволно деление и записвал края и дължината ѝ. Да се напише програма **RULER**, която извежда началото на тази летвичка, което се намира най-близко до началото на линейката..

Вход:

На първите три реда от стандартния вход се въвеждат по две цели числа, разделени с интервал – дължината и края на всяка от летвичките..

Изход

Програмата да изведе цяло число, което показва началото на летвичката, която се намира най-близко до началото на линейката.

Ограничения

Въведените числа са не по-големи от **100000** и не по-малки от **1**.

ПРИМЕР

Вход

3 9
8 12

6 15

Изход

4

Решение:

Нека променливите **x** и **x2** служат за въвеждане на двойката числа, указващи дължината и края на летвичката. За да се намери началото на всяка летвичка, извеждаме от числото показващо края на летвичката нейната дължина и получената стойност започваме в променлива **x1**. Спрямо намерената стойност за началото на летвичката прилагаме алгоритъм за намиране на минимално число.

I начин:

Без да се използва оператор за цикъл

```
#include<iostream>
using namespace std;
int main ()
{ long x, x1, x2, min;
  cin>>x>>x2;
  x1=x2-x;
  min=x1;
  if (x1<min) min=x1;
  cin>>x>>x2;
  x1=x2-x;
  if (x1<min) min=x1;
  cin>>x>>x2;
  x1=x2-x;
  if (x1<min) min=x1;
  cout<<min<<endl;
  return 0;
}
```

II начин:

Като се използва оператор за цикъл

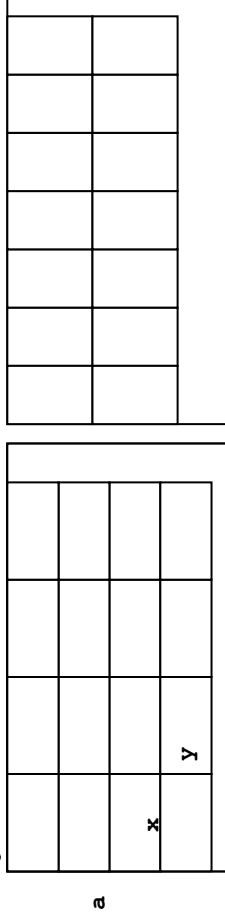
```
#include<iostream>
using namespace std;
int main ()
{ long x, x1, x2, min=100000;
  for(int i=0; i<3; i++)
  {
    cin>>x>>x2;
    x1=x2-x;
  }
```

```
if (x1<min) min=x1;
}
cout<<min<<endl;
return 0;
}
```

автор: Бистра Танева

Задача E2. Тениски

Организаторите на Есенния турнир по програмиране решили да поръчат тениски за участниците. След като венчки тениски били ушити, се оказало, че останало правоъгълно парче плат с ширина **a** и дължина **b**. Товага дизайнерите решили да разкрасят тениските като им добавят джобчета с размери **x** и **y**. За да изглеждат добре, трябвало или всички джобчета да се изрежат по дължината на плата, или всички да са по ширината – не се позволявало комбинирано изрязване.(вижте картинката)



b

Дизайнерите забелязали (както, предполагам и вие), че в двата случая се получават различен брой джобове. Те се интересуват какъв е най-големия брой джобове, който може да се получи при дадени размери на парчето плат и размерите на един джоб.

Вход:

На първия ред на стандартния вход се въвеждат четири числа **a, b, x** и **y** – размерите на парчето плат и размерите на един джоб.

Изход:

Програмата извежда на първия ред на стандартния изход едно цяло число – максималния брой джобове, а на втория площта на останалото парче плат като цяло число.

ПРИМЕР

Вход	Изход
6 8 2 3	8 0

Решение:**Необходимы величины:**

```

int a, b; - за размерите на плата;
int x, y; - за размерите на джоба;
int brh; - за броя джобове при хоризонтално разкрояване;
int brv; - за броя джобове при вертикално разкрояване;
int sh, sv; - за площта на оставащата част съответно при хоризонтално и
вертикално разкрояване;

```

Входни данни:

Въвеждат се числата a , b , x и y .

Пресмята се броя джобове при хоризонтално разкрояване, като се има предвид, че броя джобове, които ще се получат от един хоризонтален ред е a/x , а броя на хоризонталните редове е b/y :

```
brh = (a/x) * (b/y);
```

Пресмята се оставащата площ при хоризонтално разкрояване като от площта на плата се извади площта на изрязаните джобове:

```
sh = a*b - brh * (x*y);
```

Пресмята се броя джобове и оставащата площ при вертикално разкрояване:

```
brv = (a/y) * (b/x);
```

```
sv = a*b - brv * (x*y);
```

Определя се кое от двете разкроявания е максимално и се извежда съответстващият брой и съответстващата площ.

```
if (brv > brh) cout << brv << endl << sv << endl;
```

```
else cout << brh << endl << sh << endl;
```

Окончателната програма – решение на задачата:

```

#include <iostream>
using namespace std;
int main()
{
    int a, b, x, y, brh, brv, sh, sv;
    cin >> a >> b >> x >> y;
    brh = (a/x) * (b/y);
    sh = a*b - brh * (x*y);
    brv = (a/y) * (b/x);
    sv = a*b - brv * (x*y);
    if (brv > brh) cout << brv << endl << sv << endl;
    else cout << brh << endl << sh << endl;
    return 0;
}

```

Задача Е3. Спестовник

Майката на Гошко го изпратила да купи носни кърпички от близкия супермаркет. Разглеждайки цените на различните опаковки Гошко установил следното: един пакет носни кърпички струва **12** ст. един стек (**8** пакета) - **82** ст., а една кутия (**40** стека) - **30.5** лв. Гошко трябвало да купи **N** пакета и забелязал, че големите опаковки са по-изгодни. Той се замислил как да направи най-изгодната покупка. Напишете програма **SAVER**, която намира най-евтиния начин за закупуване на даден брой пакети носни кърпички.

Вход

На входа се въвежда естествено число **N** ($0 < N < 1000000$) за броя пакети носни кърпи.

Изход

На изхода се извеждат **3 неотрицателни числа** разделени с по един интервал показващи съответно- брой кутии брой стекове, брой пакети.

Примери:**Вход:**

```
14
```

Изход:

```
0 1 6
```

Вход:

```
310
```

Изход:

```
1 0 0
```

Решение:

Задачата е много подходяща за въвеждане на „новациите“ в „състезателното програмиране“. За решението не са нужни никакви специални знания освен „разклонение“. Независимо от простотата чрез нея могат да се „хванат в грешка“ и много добри ученици.

```

#include <iostream>
using namespace std;
int main()

```

```

{
    int a=1, /*пакет*/ b=8, /*стек*/ c=320; /*кутия*/
    long br1,ol,n,p,q;
    float c1,c8,c320/*цены на пакет,стек,кутия*/;
    cin>>n;
    c1=0.12;c8=0.82;c320=30.5;
    p=int(c8/c1)+1;q=int(c320/c8)+1;
    br1=n/c;ol=n%c;
    if(ol+(40-q)*b+(8-p)*a>=c)
        cout<<br1+1<<" "<<0<<" "<<0<<endl;
    else
    {
        cout<<br1;br1=ol/b;ol=ol%b;
        if (ol+(8-p)*a>=b)
            cout<<" "<<br1+1<<" "<<0<<endl;
        else cout<<" "<<br1<<" "<<ol<<endl;
    }
}

```

автор: Теодосу Теодосиев