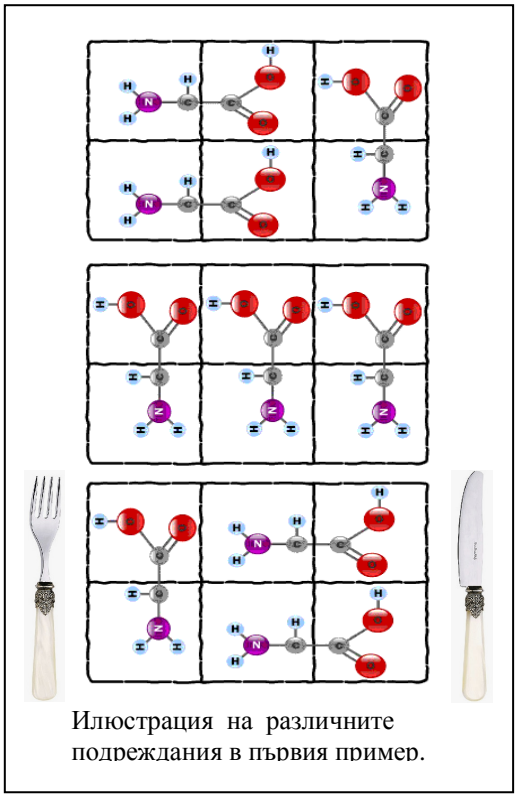
ТЕМА ЗА ГРУПА А (11-12 КЛАС)

Задача А1. АМИНОСУПА, автор Петър Иванов, по идея на Светослав Колев

Въпреки предложенията, от началото на миналия век, да се обяви Американското Патентно Ведомство за ненужно, защото „всичко вече е изобретено“, страстни учени са убедени в противното. В търсене на нови видове изчислителна техника се раждат странни и (понякога) гениални идеи. Група от млади биолози подготвя сериозен модел на вероятностна изчислителна машина, наречена “Аминокиселинна чорба”, състояща се от хаотично поставени аминокиселини. На теория, всяка аминокиселина може да се представи като правоъгълно парче с размери 1 x 2. С тези парчета трябва плътно да се покрие правоъгълна мрежа от квадратчета с размер N x M (без парчетата да се припокриват). Всяко парче може да бъде поставено в едно от двете възможни положения – по дължина или по ширина на мрежата, като не се прави разлика между двата края на парчето. Понеже процесът на изчисление на вероятностната машина е дълъг и сложен и се „разпространява“ по едното измерение на мрежата, нужна е голяма стойност на дължината N, за разлика от стойността на ширината M. За вероятностните изчисления е нужна комбинаторика, разчитаща на броя на различните аминокиселинни подреждания, които могат да се образуват върху решетката под влияние на околната среда.



Илюстрация на различните подреждания в първия пример.

Обикновено експериментаторите се интересуват от „порядъка“ на резултатите, но в случая високо се цени Вашата способност за точни изчисления. Напишете програма **aminosoup**, която извежда остатъка, получен когато броя на различните покривания на мрежата с аминокиселини се раздели на 602214179.

Вход. На единствения ред на стандартния вход са зададени естествените числа N и M.

Изход. На единствения ред на стандартния изход изведете броя на различните запълвания на мрежа с размери N x M, които се получават при плътно подреждане на правоъгълници с размери 1 x 2.

Ограничения

$1 < N < 10^{16}$, $0 < M < 7$.

ПРИМЕР 1	**ПРИМЕР 2**
Вход:	**Вход:**
`2 3`	`6 3`
Изход:	**Изход:**
`3`	`41`

РЕШЕНИЕ

Ще представим мрежата от квадратчета с матрица, а аминокиселините, запълващи мрежата, с двойки съседни в ред или стълб елементи. Понеже правоъгълниците са с размер 1x2, то един правоъгълник не може да присъства в повече от два реда на матрицата. Ще разделим задачата на две части.

Първа част. Да предположим, че сме запълнили изцяло първите `k-1` реда на матрицата, като сме оставили `k`-тия ред запълнен частично, а следващите редове не сме запълнили изобщо. Интересува ни, за всяко възможно частично запълване на елементите в ред `k-1`, по колко различни начина можем да допълним този ред, оставяйки `k`-тия запълнен частично, а следващите редове – незапълнени. Тази подзадача може да се реши, използвайки техниката “динамично оптимиране” или рекурсивно изчерпване на възможните дозапълвания. Ще кодираме запълванията на ред с дължина M с естествено число в интервала $[0,t=2^M)$, всеки бит на което ще съответства на запълнеността на съответната клетка от реда. В резултат получаваме матрицата на преходите `A`, която ни показва по колко начина можем да преминем от фиксирано частично запълнен ред към следващ ред, чието частично запълване също сме фиксирали – броят на преходите от запълването кодирано с `X` към частично запълване на следващия ред, кодирано с `Y` е `A[X][Y]`.

Във втората част на решението ще намерим броя на начините за запълване на цялата матрица. Нека от запълване `X` можем да преминем към частично запълване на следващия ред, кодирано с `Y`, по `A[X][Y]` начина, а от запълване, кодирано с `Y`, преминаваме към запълване, кодирано с `Z`, по `A[Y][Z]` начина. Поради независимостта на тези две запълванията следва, че можем да запълним последните два реда по `A[X][Y]*A[Y][Z]` начина. При това последният ред ще е частично запълнен и запълването му ще е кодирано с `Z`. По индукция следва формула за N последователни реда. Понеже не ни интересуват конкретните междинни запълвания на редовете (те могат да бъдат произволни), разглеждаме всички възможни преходи, което съответства на повдигане на матрицата на преходите на квадрат. За да получим броя на всички запълвания, трябва да повдигнем матрицата на N-та степен, използвайки алгоритъм за бързо повдигане на матрица на степен. Търсеният брой е съдържанието на клетката `B[t-1][t-1]` = A^n`[t-1][t-1]` – броят на преходите от изцяло запълнен първи ред, към изцяло запълнен последен ред.

Възможни подходи за частично решаване на задачата (поради висока сложност на алгоритмите) са също така пълното изчерпване и динамично оптимиране за всички редове без умножение на матрици.

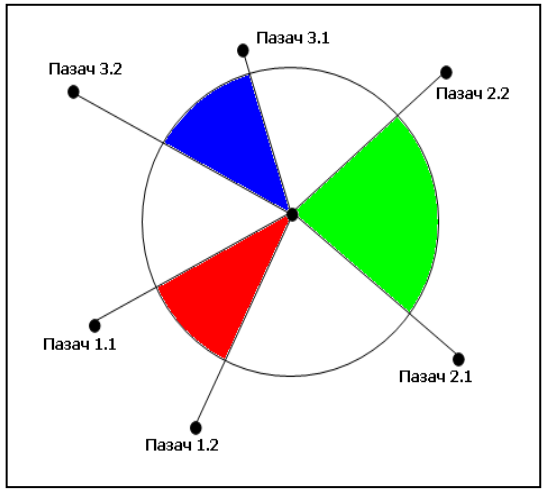
```
#include <iostream>
#include <memory.h>
using namespace std;
const int MAX_M = 6;
const int MOD = 602214179;
typedef long long type;
typedef type matrix[1<<MAX_M][1<<MAX_M];
type n; int m, t; matrix A, B;
void matrix_init ()
{ int i, j, k;
  int dp[MAX_M];
  for(i=0;i<t;i++) {
    for(j=0;j<t;j++) {
    memset(dp,0,sizeof(dp));
    for (k=0; k<m; k++) {
        if (j&(1<<k)) {
          if (i&(1<<k))
              if (k&&(j&(1<<(k-1)))&&(i&(1<<(k-1))))
                dp[k] += k>1?dp[k-2]:1;
            else dp[k] += k?dp[k-1]:1;
      }
      else if (i&(1<<k))dp[k] += k?dp[k-1]:1;
      }
        A[i][j] = dp[m-1];
    }
  }
}
inline void add_mod(type &a,type b) {a=(a+b)%MOD;}
// умножение на две матрици txt -> O(t^3)
void matrix_mult (matrix C, matrix D)
{ int i, j, k;
  matrix ans;
  memset(ans,0,sizeof(ans));
  for (i=0; i<t; i++)
    for (j=0; j<t; j++)
        for (k=0; k<t; k++)
          add_mod(ans[i][j],C[i][k]*D[k][j]);
  memcpy(C,ans,sizeof(ans));
}
// повдигане на матрица на степен p -> O(log(p))
void matrix_power (type p)
{ if (p==1) memcpy(B,A,sizeof(A));
  else {
    matrix_power (p>>1);
    matrix_mult(B,B);
    if (p&1) matrix_mult(B,A);
  }
}
int main ()
{  cin >> n >> m;
  t = 1<<m;
  matrix_init();
  matrix_power(n);
  cout << B[t-1][t-1] << "\n";
  return 0;
}
```

Задача А2. ОХРАНА, автор Бойко Банчев

Голям кръгъл склад се охранява от разположени отвън три двойки пазачи, обърнати с лице към средата му. Всяка двойка отговаря за един определен участък от стената. Първият пазач във всяка двойка следи (без непременно да я вижда цялостно) частта от стената надясно от точката пред себе си до точката на стената, намираща се пред втория пазач. Партньорът му следи същия участък – от право пред себе си наляво до точката пред първия (виж фигурата). Напишете програма **guard**, която проверява дали по този начин се следи цялата стена.

Вход. На стандартния вход са зададени двойки цели числа – координати на зададените точки. На първия ред са координатите на центъра на склада, а на следващите три – тези на пазачите, по една двойка на ред – за първия и втория пазач във всяка двойка, съответно. Всички координати са в интервала [-150,150].

Изход. На ред от стандартния изход се отпечатва цяло число – наблюдаваната част от стената в проценти, закръглена до най-близката цяла стойност. Ако наблюдаваната част е еднакво близка до две цели числа, вземете по-голямото.



ПРИМЕР

Вход:	**Изход:**
0 0 100 -1 0 100 0 120 -50 -95 -60 -90 90 10	100

РЕШЕНИЕ

За всяка от двойките пазачи, позициите им и центърът на склада определят дъга от окръжност, чийто радиус е без значение. По дадените точки намираме краищата на дъгите в ъглови стойности. Търсената покрита част от стената е мярката на обединението на трите дъги, отнесена към цялата окръжност.

Намирането на тази мярка непосредствено е неудобно, понеже обединението на две дъги се състои от една или две дъги, която или които после трябва да обединим с третата дъга (и толкова по-трудно, ако дъгите са повече от три!). Затова е добре да приложим метода на включване и изключване. В случая това значи да намерим числото $a_1+a_2+a_3-(a_{12}+a_{23}+a_{31})+a_{123}$, където a_1, a_2 и a_3 са мерките на самите дъги, a_{12}, a_{23} и a_{31} са мерките на сеченията (общите части) на a_1 с a_2, a_2 с a_3 и a_3 с a_1, а a_{123} е мярката на сечението (общата част) на трите дъги заедно.

Как намираме сеченията? Трябва да забележим, че сечение на две дъги също може да се състои от две дъги, но това става само тогава, когато двете дъги покриват цялата окръжност, а значи и отговорът е известен без нужда от други пресмятания. Затова сечения и мерки („дължини“ в ъглова мярка) на дъги търсим само когато всеки две дъги или нямат обща част, или тя е единствена дъга.

За всеки две дъги проверяваме вида на общата им част – 0, 1 или 2 дъги, и постъпваме съответно. Проверката за вида на общата им част, както и намирането на общата част, когато е нужно, става с помощта на една по-проста проверка: дали дадена ъглова стойност се намира между две други върху окръжността, т.е. дали точка под даден ъгъл лежи на дадена дъга.

При пресмятанията с ъглови стойности – събирания, изваждания и сравнения – трябва да се внимава във връзка с факта, че дъговите интервали имат граници в циклична, а не просто линейна числова област. В това отношение лесно се правят грешки.

```
#include <iostream>
#include <cmath>
#include <utility>
using namespace std;
#define DPI (2*3.1415926536)
#define EPS .0001
typedef pair<double,double> prd;
inline bool lt(double a, double b)  {return a<b-EPS;}
inline bool eq(double a, double b)
{return fabs(a-b)<=EPS;}
inline void normalize(prd & a)
{if (a.first>a.second)  a.second += DPI;}
bool onarc(double a, prd arc)  {
  if (a>=DPI)  a -= DPI;
  return lt(arc.first,a) && lt(a,arc.second)
      || lt(arc.first,a+DPI) && lt(a+DPI,arc.second);
}
double xsect(prd a1, prd a2, prd & x)  {
  bool on11,on12,on21,on22;
  on12 = onarc(a1.first,a2); on22 = onarc(a1.second,a2);
  on11 = onarc(a2.first,a1); on21 = onarc(a2.second,a1);
  if (!(on11 || on12 || on21 || on22 ||
        eq(a1.first,a2.first) && eq(a1.second,a2.second)))
    return 0.;
  else if (on11 && on12 && on21 && on22) return -1.;
  x.first = on12 ? a1.first : a2.first;
  x.second = on21 ? a2.second : a1.second;
  if (x.first>=DPI)  x.first -= DPI;
  while (x.second-x.first>=DPI)  x.second -= DPI;
  if (eq(x.second,x.first))  return 0.;
  normalize(x);
  return x.second-x.first;
}
int main()  {
  prd o,g1,g2,a[3],x;
  double s,s1,s2,s3;
  int i,p;
  cin >> o.first >> o.second;
  for (i=0; i<3; ++i)  {
    cin >> g1.first >> g1.second >> g2.first >> g2.second;
    a[i].first=atan2(g1.second-o.second,g1.first-o.first);
    a[i].second=atan2(g2.second-o.second,g2.first-o.first);
    if (a[i].first<0.)  a[i].first += DPI;
    if (a[i].second<0.)  a[i].second += DPI;
    normalize(a[i]);
  }
  s1 = xsect(a[1],a[2],x); s2 = xsect(a[2],a[0],x);
  s3 = xsect(a[0],a[1],x);
  if (s1<0 || s2<0 || s3<0)  p = 100;
  else  { s = a[0].second-a[0].first + a[1].second -
          a[1].first + a[2].second-a[2].first - (s1+s2+s3);
  if (s1>0. && s2>0. && s3>0.)  s += xsect(x,a[2],x);
        p = (int)(.5+s*100./DPI);
      }
  cout << p << endl;
  return 0;
}
```

Задача А3. ГЕНЕРАТОР НА ОБИКНОВЕНИ ДРОБИ, автор Младен Манев

Петър е ученик в шести клас. Задачите от групи E и D на всички състезания по информатика вече не са никакъв проблем за него. Ето защо той започна сам да си измисля задачи, че даже и да ги решава. Най-новото му творение е програма за генериране на обикновени дроби. Ето как работи тя. Петър въвежда несъкратима обикновена дроб $x = p/q$ (p и q са положителни цели числа). Генерирането на нови дроби се извършва чрез командите ***L*** или ***R***. При команда ***L*** програмата извежда несъкратима дроб, равна на $2x+1$, а при ***R*** – несъкратима дроб, равна на $\frac{x}{x+2}$. При всяко следващо въвеждане на команда ***L*** или ***R***, за генерирането на нова дроб програмата използва последната получена. Например, след въвеждане на $\frac{2}{3}$, ***R***, ***L***, ***L*** и ***R***, тя извежда последователно дробите $\frac{1}{4}$, $\frac{3}{2}$, $\frac{4}{1}$ и $\frac{2}{3}$. Сега Петър иска да разбере колко пъти най-малко трябва да въведе ***L*** или ***R***, за да може програмата му да генерира дробта, която е въвел. Помогнете на Петър, като напишете програма **gen**, която решава новата задача.

Вход. На един ред на стандартния вход са зададени две цели положителни числа – числителят p и знаменателят q на въведената от Петър несъкратима дроб.

Изход. На един ред на стандартния изход програмата трябва да изведе най-малкия брой команди, необходими за да се генерира зададената дроб. Ако въведената от Петър дроб не може да бъде генерирана, програмата трябва да изведе 0.

Ограничения. $1 \le p \le 10\,000\,000$, $1 \le q \le 10\,000\,000$

ПРИМЕР

Вход	**Изход**
2 3	4

РЕШЕНИЕ

Нека $x=\frac{p}{q}$ (p и q са взаимно прости цели положителни числа). Тогава $2x+1=\frac{2p+q}{q}$ и $\frac{x}{x+2}=\frac{p}{p+2q}$. Ако може да съкратим някоя от тези дроби, то може да я съкратим само на 2, защото

$(2p+q,q)=(2p,q)\le 2(p,q)=2$ и $(p,p+2q)=(p,2q)\le 2(p,q)=2$.

Следователно сумата на числителя и знаменателя в новополучените несъкратими дроби ще е или $2(p+q)$, или $p+q$. Тъй като програмата на Петър трябва да генерира въведеното от него число, то при всяко въвеждане на ***L*** или ***R*** трябва да се получава несъкратима дроб със сбор на числителя и знаменателя $p+q$. За p и q има две възможности:

1. И двете числа са нечетни. Тогава дробите $\frac{2p+q}{q}$ и $\frac{p}{p+2q}$ са несъкратими и програмата на Петър няма да може да генерира въведената от него дроб.

2. Двете числа са с различна четност. Ако p е четно число, а q – нечетно, то дробта $\frac{2p+q}{q}$ е несъкратима, а дробта $\frac{p}{p+2q}$ – съкратима. В този случай, за да не се промени сумата на числителя и знаменателя, Петър трябва да въведе ***R***. Ако p е нечетно число, а q – четно, то дробта $\frac{2p+q}{q}$ е съкратима, а дробта $\frac{p}{p+2q}$ – несъкратима. В този случай, за да не се промени сумата на числителя и знаменателя, Петър трябва да въведе ***L***. За новата дроб отново числителят и знаменателят са цели числа с различна четност. Числото $p+q$ може да се представи като сбор на две положителни цели числа по $p+q-1$ начина. Това означава, че ако след $p+q-1$ въвеждания на ***L*** или ***R***, отчитайки горните разсъждения, началната дроб не е била генерирана, то тя няма да може да бъде изведена от програмата на Петър.

```
#include<iostream>
using namespace std;
int main()
{  int p,q,p1,q1,br;
   cin>>p>>q;
   if (p%2==1&&q%2== 1)
   {  cout<<0<<endl; return 0; }
    p1=p; q1=q; br=0;
    do
    { if (p1%2==0) {p1=p1/2;q1=p1+q1; }
      else {q1=q1/2;p1=p1+q1; }
      br++;
    }while(p1!=p&&br<p+q);
    if (br==p+q) cout<<0<< endl;
    else cout<<br<< endl;
    return 0;
}
```

ТЕМА ЗА ГРУПА В (9-10 КЛАС)

Задача В1. ПРИСВОЯВАНЕ, автор Явор Никифоров

Когато в програма напишем `a=b` (или `a:=b`), очакваме, че съдържанието на променливата `b` ще се копира на мястото на досегашното съдържание на `a`. Възможна е и друга концепция за присвояване: в резултат на операцията `p=q` променливата `p` сочи към това място в паметта, където се съхранява стойността на променливата `q`. При такова присвояване, когато на някоя от променливите, сочещи към място в паметта, бъде дадена нова стойност, всички други променливи, сочещи към това място получават новата стойност.

Задачата е да напишете интерпретатор **`inter`** за език, в който съществуват и двете форми на присвояване. В този език всички променливи имат числови стойности, а имената им също са числа.

Програмният фрагмент, който трябва да интерпретирате съдържа 3 вида команди:

`C x y` – където `x` и `y` са имена на променливи, означава: в паметта, където сочи `x` да бъде копирано това, към което сочи `y` (обичайно присвояване).

`R x y` – където `x` и `y` също са имена (номера) на две променливи, означава: променливата с име `x` да започне да сочи там, където сочи променливата `y`.

`N x` – където x е име на променлива, означава, че в паметта се заделя място, в него се поставя резултатът от функцията `getnextint`, а променливата `x` започва да сочи към това място в паметта. Функцията `getnextint` винаги връща с 1 повече от предишния път, когато е била извикана, започвайки от 0. Преди да започне интерпретацията на кода, зададен на входа, всяка от променливите е имала стойност равна на номера си, в резултат на съответен брой извиквания на `getnextint` (тази част от кода няма да бъде включена във фрагмента).

Вход. На първия ред на стандартния вход са зададени броят P на променливите и броят M на редовете на програмата ($P < 500\,000$, имената на променливите са числата от 0 до $P - 1$, $M < 10\,000\,000$). На всеки от следващите M реда ще бъде зададена по една команда.

Изход. На един ред на стандартния изход програма трябва да изведе стойностите, които ще имат след изпълнение на програмата, променливите 0,1,..., $P - 1$, в този ред.

Ограничения. Тази задача трябва да решите с ограничение на паметта от 10 MB.

ПРИМЕР

Вход:
```
7 5
R 1 0
C 2 0
C 0 3
R 4 5
N 5
```

Пояснение: Ако проследим програмата стъпка по стъпка, то променливите ще имат стойностите:

отначало:	0 1 2 3 4 5 6
след 1 команда:	0 0 2 3 4 5 6
след 2 команди:	0 0 0 3 4 5 6
след 3 команди:	3 3 0 3 4 5 6
след 4 команди:	3 3 0 3 5 5 6
в края:	3 3 0 3 5 7 6

Изход:
```
3 3 0 3 5 7 6
```

РЕШЕНИЕ

От условието е очевидно, че трябва да бъдат разделени понятието променлива (полето от паметта къде сочи тя) и понятието стойност (съдържанието на полето към което тя сочи).

Заденото ограничение на паметта (10MB) изглежда достатъчно, тъй като променливите са не повече от 500 000. Стойностите към които може да бъде сочено, обаче биха могли да достигнат до 10 500 000 (първоначални 500 000 + 10 000 000 извиквания на команда N).В същото време 500 000 променливи не могат да сочат повече от 500 000 стойности. Веднага когато стойност остане без променлива, която да сочи към нея, тя става неизползваема – няма команда с която променлива да се насочи към нея по-късно. В такъв случай ще е удачно да разпознаваме такива полета памет и да ги освобождаваме за следващо използване – те биха могли да ни потрябват при една изненадващо дълга серия от команди N.

Времевите ограничения не биха ни позволили тривиални решения. Например, да търсим поле от паметта, към което не сочи никоя променлива. Това трябва да установяваме мигновенно, като пазим броя на променливите сочещи към всяка от стойностите, например. Ще имаме нужда също да намираме с константна скорост свободно поле. Едно решение на този въпрос е, да поддържаме някаква линейна структура със свободните полета, в която да можем да добавяме и от която да вземаме елементи. Една възможност е стек, имплементиран като свързан списък, в който всяко поле от паметта, към което сочат 0 променливи, е указател към следващо неизползвано поле от паметта. Достъпът в такъв свързан списък осъществяваме през последното добавено поле – връх на стека.

Така се очертават следните реализации на трите команди:

`C` (тривиална) – стойността към която сочи x се заменя от стойността към която сочи y.

`R` – променливата x се пренасочва към мястото, към което сочи y. Това намалява с 1 броя на променливите, сочещи към полето, където е сочела `x`. Ако броят стане 0 – освободеното поле се добавя в стека на свободните полета.

`N` – взема се свободно поле от върха на стека. Поставя се в него новата стойност, а x започва да сочи там. С евентуално освободеното от `x` поле постъпваме както при R.

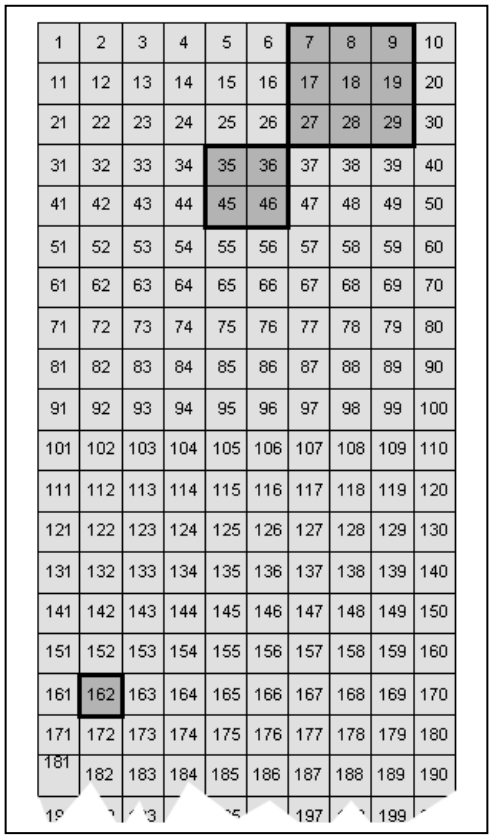
След отработването на тези команди, отпечатваме стойностите цитирани от всяка от променливите.

```
#include<stdio.h>
#define max_var 500003
using namespace std;
int memory[max_var];
int usages[max_var];
int points[max_var];
int var, orders, nextint, firstfree;
int getnextint(){
```

```
   nextint=nextint+1;
   return nextint-1;
}
int getnewfreecell(){
   int tmp=firstfree;
   firstfree=memory[firstfree];
   return tmp;
}
void justfreed(int to){
   usages[to]--;
   if(usages[to]<1){
      usages[to]=0;
      memory[to]=firstfree;
      firstfree=to;
   }
}
int main(){
   nextint=0;
   scanf("%d %d\n",&var,&orders);
   for(int i=0;i<var;i++){
      points[i]=i;
      usages[i]=1;
      memory[i]=getnextint();
   }
   char o;
   int from, to;
   firstfree=var;
   memory[var]=var+1;
   usages[var]=0;
   for(orders;orders>0;orders--){
      scanf("%c",&o);
      if(o=='N'){
         scanf("%d\n",&to);
         justfreed(points[to]);
         int ind=getnewfreecell();
         memory[ind]=getnextint();
         usages[ind]=1;
         points[to]=ind;
      }
      if(o=='C'){
         scanf("%d %d\n",&to,&from);
         memory[points[to]]=memory[points[from]];
      }
      if(o=='R'){
         scanf("%d %d\n",&to,&from);
         justfreed(points[to]);
         points[to]=points[from];
         usages[points[from]]++;
      }
   }
   for(int j=0;j<var;j++) {
      if(j!=0)printf(" ");
      printf("%d",memory[points[j]]);
   }
   printf("\n");
   return 0;
}
```

Задача B2. СУМИ В КВАДРАТИ, автор Павлин Пеев

Естествените числа са записани върху мрежа от квадратчета с ширина 10, началото на която е показано вдясно. Нека е зададено цяло положително число *S*. Искаме да намерим такъв квадрат в мрежата, сумата на числата в който е точно *S*. Разбира се, такъв квадрат със страна 1 съществува и това е единичното квадратче, в което е записано самото *S*. Ние търсим квадрат, чийто горен ляв ъгъл е колкото може по-нагоре и колкото може по-наляво в мрежата, с този приоритет –„нагоре“ е по-важно от „наляво“.



Нека, например, *S*=162. На чертежа са показани три квадрата, сумата от числата в които е 162: единият е самото число 162, вторият е с горен ляв ъгъл 35 и третият – с горен ляв ъгъл 7. Именно последният е целта на тази задача.

Напишете програма **sqsum**, която открива търсения квадрат и извежда числото в горния му ляв ъгъл.

Вход. На единствения ред на стандартния вход е зададено число *S*, не по-голямо от 1000000.

Изход. Програмата трябва да изведе на стандартния изход един ред с цялото число, записано в горния ляв ъгъл на най-горния най-ляв квадрат в мрежата със сума от числата в него равна на *S*.

ПРИМЕР 1	**ПРИМЕР 2**
Вход:	**Вход:**
162	15
Изход:	**Изход:**
7	15

РЕШЕНИЕ

Задачата допуска различни подходи за решаване, които касаят подходящо търсене в мрежата. Да пресметнем, например, сумата S на числата в квадрат с горен ляв ъгъл n и страна a.

$$S = n + (n+1) + (n+2) + \ldots + (n+a-1) +$$
$$+ (n+10) + (n+11) + (n+12) + \ldots + (n+10+a-1) + \ldots +$$
$$+ (n+10(a-1)) + (n+10(a-1)+1) \ldots + (n+10(a-1)+a-1)$$

След преобразувания, тази сума може да се запише $S = \frac{a^2(2n+11a-11)}{2}$.

Този резултат води и до кратък и бърз алгоритъм за решаване на задачата: тъй като условието изисква минимизиране на n (по-малките n са по-нагоре и по-наляво), това е равносилно на максимизиране на a, което обаче може да е най-много 10 и най-малко 1. При обхождането (намаляващо) на тези стойности за a, изчисленото n трябва да отговаря на очевидни условия: да е естествено, както и да е достатъчно наляво, така че квадратът със страна a да е в рамките на мрежата. А именно, ако последната цифра на n е нула, е възможен само кварат с $a = 1$, иначе са възможни квадратите, за които $(n \bmod 10) + a \le 11$. Първото намерено n с тези свойства ще е търсеното решение.

```
#include <iostream>
using namespace std;
long find (long s)
{  long a,n,p;
   for (a=10;a>1;a--)
   if ((2*s)%(a*a)==0)
   {  p=2*s/(a*a)+11-11*a;
      n=p>>1;
      if (p>0&&!(p&1)&&n%10&&n%10+a<=11) return n;
   }
   return s;
}
int main(void)
{  long S;
   cin>>S;
   cout<<find(S)<<endl;
   return 0;
}
```

Задача В3. ШАХМАТНИ ЦАРЕ, автор Емил Келеведжиев

Дадена е шахматна дъска с N реда и M стълба. Две клетки на дъската наричаме съседни, ако имат обща страна или общ връх. Задачата е да се поставят K шахматни царя в различни клетки на дъската така, че никои два да не са в съседни клетки. Напишете програма **king**, която намира по колко различни начина може да стане това.

Вход. На първия ред на стандартния вход са зададени трите цели цели числа N, M и K.

Изход. Програмата трябва да изведе на стандартния изход едно цяло число – броя на различните начини на поставяне на царете, според условието на задачата.

Ограничения. $0 < N < 13$, $0 < M < 13$, $0 < K < 13$.

ПРИМЕР

Вход:	**Изход:**
3 3 2	16

РЕШЕНИЕ

Разглеждаме таблица от N реда и M стълба, в която всяка клетка има стойност 1 или 0, в зависимост от наличието или отсъствието на цар в клетката. Ако разгледаме един стълб в тази таблица, ще видим, че той може да съдържа в клетките си стойностите 1 или 0 така, че няма две единици една до друга. Нека да означим с B_t, t = 1,2, ..,$c(N)$, всички възможни стълбове с това свойство. Лесно се съобразява, че техният брой $c(N)$, пресметнат като функция на N е $(N + 1)$-то число от редицата на Фибоначи: $F_0 = 1$, $F_1 = 1$, а $F_N = F_{N-1} + F_{N-2}$. Наистина, при $N = 1$ имаме $F_2 = 2$ такива стълба – един с 0 и един с 1. При $N = 2$ стълбовете са $F_3 = 3$ – (0,0), (0,1) и (1,0). За $N > 2$ броят на стълбовете с 0 в първия ред е $c(N-1) = F_N$, а броят на стълбовете с 1 в първия ред е $c(N-2) = F_{N-1}$. Следователно $c(N) = F_N + F_{N-1} = F_{N+1}$.

Разглеждаме подзадача, която се получава от дадената, като се ограничим с първите m стълба на таблицата, като последният стълб при подзадачата е идентичен със стълба B_t и освен това, броят на поставените царе е k. Означаваме броя на различните конфигурации при тази подзадача с $f(m, t, k)$. Съобразяваме, че е в сила следната рекурентна зависимост:

$$f(m+1, t, k) = \sum_r f(m, r, k - p(t)),$$

където $p(t)$ е броят на единиците в стълба B_t и сумирането се извършва по тези стойности на r, за които стълбовете B_r и B_t, когато ги поставим един до друг, нямат единици в съседни клетки.

Отбелязваме, че $f(1, t, k)$ е равно на 1 или 0, в зависимост от това дали е изпълнено равенството $p(t) = k$. Тогава е възможно да организираме последователно пресмятане на $f(m, t, k)$ при всички стойности на параметрите m, t и k, и решението на задачата се получава от сумата:

$$\sum_{t=1}^{c} f(M, t, K)$$

```
#include<cstdio>
const int M_max=15, N_max=15, K_max=15;
int M, N, K;
const int max_bands=1000;
char band[max_bands][N_max+1];
int bande[max_bands];
bool conflict[max_bands][max_bands];
long long int d[M_max+1][max_bands][K_max+1];
int c=0;

void gen_band(int p=0)
{  if(p>N)
   {  c++;
      for(int j=1;j<=N;j++) band[c][j]=band[c-1][j];
      return;
   }
   band[c][p]=0; gen_band(p+1);
   if(p>0)if(band[c][p-1]==0)
   {  band[c][p]=1; gen_band(p+1); }
}

void count_bands()
{  for(int i=0;i<c;i++)
   { bande[i]=0;
     for(int j=1;j<=N;j++) bande[i] += band[i][j];
   }
}

void compute_conflicts()
{  for(int i=1;i<c;i++) conflict[i][i]=true;
   for(int i1=0;i1<c;i1++)
   for(int i2=0;i2<i1;i2++)
   for(int j=1;j<=N;j++)
   {  if((band[i1][j]==1)&&(band[i2][j]==1))
      {  conflict[i1][i2]=conflict[i2][i1]=true; break; }
      if(j>1&&band[i1][j]==1&&band[i2][j-1]==1)
      {  conflict[i1][i2]=conflict[i2][i1]=true; break; }
      if(j<N&&band[i1][j]==1&&band[i2][j+1]==1)
      {  conflict[i1][i2]=conflict[i2][i1]=true; break; }
   }
}

int main()
{  scanf("%d%d%d",&N, &M, &K);
   gen_band(0); count_bands(); compute_conflicts();
   for(int t=0;t<c;t++)
   for(int k=0;k<=K;k++)
      d[1][t][k]=(bande[t]==k)?1:0;
   for(int j=2;j<=M;j++)
     for(int t1=0;t1<c;t1++)
        for(int k=bande[t1];k<=K;k++)
        {
           long long int s=0;
           for(int t2=0;t2<c;t2++)
           if(!conflict[t1][t2])s+=d[j-1][t2][k-bande[t1]];
           d[j][t1][k]=s;
        }
   long long int s=0;
   for(int t=0;t<c;t++) s += d[M][t][K];
   printf("%I64d\n",s);
}
```

ТЕМА ЗА ГРУПА С (7-8 КЛАС)

Задача С1. ТЪРКАЛЯНЕ НА КУБЧЕ, автор Бисерка Йовчева

Дадена е шахматна дъска (8 x 8). Дадено е и кубче, всяка стена на което е еднаква по размер с клетка на дъската, а на стените му са написани цели неотрицателни числа, не по-големи от 1000. Кубчето е поставено върху една клетка на дъската и може да се премества на съседно поле, като се претърколи през съответното ребро в основата си. При търкалянето на кубчето се сумират числата от стените, които лягат на дъската (всяко число се сумира толкова пъти, колкото пъти кубчето се окаже лежащо на тази стена). Числата, записани в основата на кубчето в началната и крайната позиция, също се добавят към сумата.Тъй като на шахматната дъска са подредени фигури, кубчето не може да се претърколи на поле, в което има фигури. Да се напише програма **cube**, която намира такъв път за движение на кубчето между две зададени полета на дъската, при който то да направи най-малко претъркалвания.

Вход. На първия ред на стандартния вход, разделени с интервали, са зададени началното и крайното поле, както и шестте числа, изписани съответно на предната, задната, горната, дясната, долната и лявата стена на кубчето (позициите съответстват на началния момент, гледани откъм долната част на дъската). Координатите на полетата се задават в стандартната шахматна нотация (клоните са означени с латинските букви от *a* до *h* отляво надясно, а редовете – от 1 до 8 отдолу нагоре). Началното и крайното поле са различни и върху тях няма фигури. На следващия ред на стандартния вход е зададено цяло число N – брой на фигурите, разположени на дъската ($0 \le N \le 40$). Следват N реда с координатите на N–те полетата с фигури, зададени по описания начин и разделени с интервали.

Изход. На стандартния изход програмата трябва да изведе сумата, която се получава в резултат на сумирането на числата от стените, на които е лягало кубчето при претъркалването си. Ако кубчето може да достигне от едното до другото поле

по няколко различни начина, да се изведе най-малката сума, която се получава при неговото движение.

ПРИМЕР

Вход:	Изход:
e2 e6 0 8 1 2 1 1 2 e3 d4	13

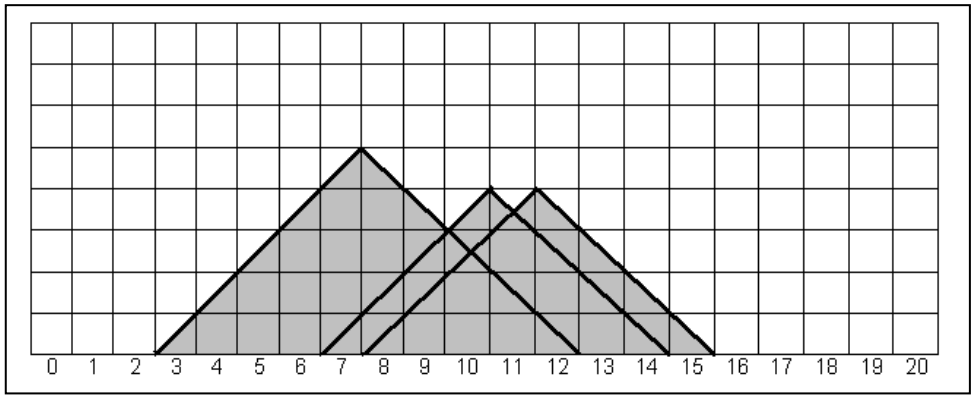
РЕШЕНИЕ

```
#include<iostream>
#include<queue>
#include<string>
using namespace std;
enum DIR {UP=0, DOWN, LEFT, RIGHT };
DIR opposite[4] = {DOWN,UP,RIGHT,LEFT };
const int dx[] = {-1,+1,0,0};
const int dy[] = {0,0,-1,+1};
const int MAX_VALUE = 8*8*1002 + 1;
struct cube {
   int front,back,top,right,bottom,left;
   int move (DIR dir) {
   int x=bottom;
   if (dir==UP)
   { bottom=back; back=top; top=front; front=x; return x; }
   if (dir==DOWN)
   { bottom=front; front=top; top=back; back=x; return x; }
   if (dir==LEFT)
   { bottom=left; left=top; top=right; right=x; return x; }
   if (dir==RIGHT)
   { bottom=right; right=top; top=left; left=x; return x; }
  }
  void read () {
   cin >> front >> back >> top >> right >> bottom >> left;
  }
};
struct cell
{
   int x, y;
   cell () {}
   cell (int _x, int _y) : x(_x), y(_y) {}
   int operator ==(cell c) { return (x==c.x && y==c.y); }
};
int a[10][10], u[10][10];
cube starting_cube;
cell starting_cell,ending_cell;
void bfs(cell c)
{
   cell k, w; queue<cell> q;  int i;
   u[c.x][c.y] = 1; q.push(c);
   while(!q.empty())
   {  k = q.front(); q.pop();
      if (k==ending_cell) break;
      for (i=0; i<4; i++) {
         w = cell(k.x+dx[i], k.y+dy[i]);
         if(a[w.x][w.y] && !u[w.x][w.y])
         {  u[w.x][w.y] = u[k.x][k.y]+1; q.push(w); }
      }
   }
}
void read()
{  string s; int i, j, n;
   for(i=1;i<=8;i++)
      for(j=1;j<=8;j++) a[i][j]=1;
   cin>>s;
   starting_cell = cell (7-(s[1]-'1')+1, s[0]-'a'+1);
   cin>>s;
   ending_cell = cell (7-(s[1]-'1')+1, s[0]-'a'+1);
   starting_cube.read();
   cin>>n;
   for(i=1;i<=n;i++)
   { cin>>s; a[7-(s[1]-'1')+1][s[0]-'a'+1]=0; }
}
void print_u()
{  int i,j;
   for(i=1;i<=8;i++)
   {  for(j=1;j<=8;j++) cout<<u[i][j]<<' '; cout<<endl; }
}
int restore(DIR cr_dir, cell cr_cell,cube &resulting_cube)
{  int i, cr_sum, min_sum=MAX_VALUE;
   cube cr_cube; cell pr_cell;
   if (cr_cell==starting_cell) {
      min_sum = starting_cube.bottom;
      resulting_cube = starting_cube;
      resulting_cube.move (cr_dir);
   }
   else { for (i=0; i<4; i++) {
      pr_cell=cell(cr_cell.x+dx[opposite[i]],
```

```
     cr_cell.y+dy[opposite[i]]);
     if(u[pr_cell.x][pr_cell.y]+1==u[cr_cell.x][cr_cell.y])
     {
      cr_sum=restore((DIR)i,pr_cell,cr_cube)+cr_cube.bottom;
      if (curr_sum < min_sum)
      { min_sum=curr_sum; resulting_cube=curr_cube; }
     }
    }
    resulting_cube.move(curr_dir);
   }
   return min_sum;
}
int main()
{  cube resulting_cube;
   read();
   bfs(starting_cell);
   cout << restore(UP,ending_cell,resulting_cube) << endl;
   return 0;
}
```

Задача С2. ДОЛИНИ, автор Красимир Манев

Дадени са N равнобедрени правоъгълни триъгълника, които са поставени върху хоризонталната числова ос. Дължината на основата на всеки такъв триъгълник е четно число, а левият и десният й край лежат на целочислени точки на оста. При това, частите от бедрата на триъгълниците, които не са закрити от други триъгълници, описват интересна начупена линия, като силует на планина. Точките от силуета, за които линията и от двете им страни върви нагоре можем да наречем долини. Напишете програма **valley**, която по зададени триъгълници намира броя на долините в получения силует.



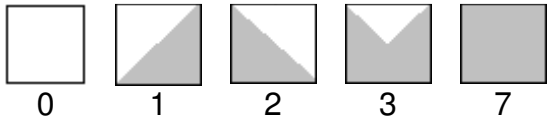
Вход. На първия ред на стандартния вход ще бъде зададен броят N на триъгълниците ($3 \le N \le 1000$). Всеки от следващите N реда съдържа описание на един от триъгълниците с две цели числа – точката B от числовата ос ($0 \le B \le 2000$), в която е левият край на триъгълника и дължината L на основата му ($2 \le L \le 2000$).

Изход. На стандартния изход програмата трябва да изведе броя на долините в силуета.

ПРИМЕР

Вход:	**Изход:**
3 8 10 8 8 7 8	2

РЕШЕНИЕ



Задачи, при които частта от равнината, в която са разположени зададените обекти не е много голяма, можем да решаваме с техника, която ще наричаме „растерна“ – за всяко квадратче със страна единица отделяме поле в паметта, в което да отразяваме случващото се със съответното квадратче. Както се вижда от Фигурата, при поставяне на оцветени равнобедрени правоъгълни триъгълници върху хоризонталната ос, могат да се получат 5 вида оцветяване на квадратчетата, които ще кодираме както е показано в таблицата. Да започнем да поставяме триъгълниците по реда, по който са зададени във входа и да отрязяваме поставянето в растера. Триъ-гълник с параметри B и $L = 2.K$ ще засегне редовете с номера 0, 1, …, $K - 1$, като в реда с номер j ще трябва да бъдат проверени стълбовете с номера от $B + J$ до $B + L - 1 - J$. В първия и последния стълб от новия триъгълник ще дойде оцвтяване от тип 1 и 2 съответно, а за всички останали – от тип 7. При наслагване на оцветяване от тип 7 върху друго – винаги получаваме в растера тип 7. Лесно се проверява, че когато наслагваме оцветяване от тип 1 или 2 върху друго, например `x`, тогава резултатът е `x|1` или `x|2`, съответно.

След като сме нанесли върху растера всички триъгълици остава да се „разходим“ внимателно по силуета и да преброим получилите се долини. Долина имаме в квадратче на растера с оцветяване 3 или при две съседни квадратчета в един ред – лявото оцветено с 2, а дясното – с 1. Допълнителните променливи `left` и `right`, в които съхраняваме левия и десния край на силуета ни помагат по-бързо да го обходим.

```
#include <stdio.h>
char a[1001][2001]={0};
int main()
{
    int N,beg,len,i,j,k,br;
```

```
    int left=1001,right=0;

    scanf("%d",&N);
    for(i=1;i<=N;i++)
    {
       scanf("%d %d",&beg,&len);
       if(beg<left)left=beg;
       if(beg+len-1>right) right=beg+len-1;
       for(j=0;j<len/2;j++)
       {  a[j][beg+j]|=1;a[j][beg+len-j-1]|=2;
          for(k=beg+j+1;k<beg+len-j-1;k++)a[j][k]=7;
       }
    }
    j=1;br=0;
    for(i=left+1;i<=right;i++)
    {
      switch(a[j][i]) {
         case 1: if(a[j][i-1]==2) br++;
                 if(a[j+1][i+1]!=0)j++; break;
         case 2: if(j>0&&a[j][i+1]==0)j--; break;
         case 3: br++; if(a[j+1][i+1]!=0)j++; break;
       }
    }
    printf("%d\n",br);
    return 0;
}
```

Сложността на този алгоритъм, за съжаление, зависи от сумата на лицата на зададените триъгълници, максималната стойност на която, при ограниченията на задачата, може да достигне до 1 000 000 000. Струва си да се опитаме да решим задачата с алгоритъм, сложността на който не зависи от големината на лицата на зададените триъгълници. За целта, нека първо сортираме параметрите на зададените триъгълници в нарастващ ред на позицията на началото, а при равна такава позиция – в намаляващ ред на дължината на основата. Сега може да започнем да сканираме силуета.

Очевидно е, че заради начина по който сортирахме триъгълниците, за да се получи долина трябва левия склон на един триъгълник да пресече десния склон на друг триъгълник, разположен вляво от първия. Затова фиксираме един триъгълник, започвайки с първия в сортираната последователност, с ляв край `BB` и дължина на основата `LL`. Всички триъгълници, за които `B[i]==BB||B[i]+L[i]<=BB+LL` пропускаме, защото фиксираният триъгълник ги покрива изцяло. Когато достигнем до триъгълник, който не се покрива изцяло от фиксирания, остава да проверим условието двата им склона да се пресичат, т.е. `B[i]<=BB+LL`. Ако условието е изпълнено, отброяваме долина. Заменяме `BB` с `B[i]` и `LL` с `L[i]`, след което продължаваме търсенето.

```
#include <stdio.h>
int B[1000],L[1000];
int main()
{  int N,i,j,t,BB,LL,br=0;
   scanf("%d",&N);
   for(i=1;i<=N;i++)
      scanf("%d %d",&B[i],&L[i]);
   for(i=N-1;i>=1;i--)
      for(j=1;j<=i;j++)
         if(B[j]>B[j+1]||B[j]==B[j+1]&&L[j]<L[j+1])
         {  t=B[j];B[j]=B[j+1];B[j+1]=t;
            t=L[j];L[j]=L[j+1];L[j+1]=t;  }
   BB=B[1];LL=L[1];
   for(i=2;i<=N;i++)
   {  if(B[i]==BB||B[i]+L[i]<=BB+LL) continue;
      if(B[i]<=LL+BB) br++;
      BB=B[i];LL=L[i]; }
    printf("%d\n",br);
    return 0;
}
```

Задача С3. ДЕЛИТЕЛИ, автор Стоян Капралов

Дадени са *N* цели положителни числа. Напишете програма **`divcnt`**, която намира броя на положителните делители за всяко от дадените числа.

Вход. На първия ред на стандартния вход ще бъде зададено числото *N*, а на втория, разделени с интервали – *N*-те числа, за всяко от които трябва да се намери броят на делителите му.

Изход. Програмата трябва да изведе един ред на стандартния изход *N* числа, разделени с интервали: за всяко от зададените във входа числа числа – броя на неговите положителни делители.

Ограничения. $1 \le N \le 100$. Дадените числа са цели и са от интервала $[1,10^9]$.

ПРИМЕР

Вход:	**Изход:**
4 3 6 10 16	2 4 4 5

РЕШЕНИЕ

Лема 1. Ако $x = p_1^{k_1} \; p_2^{k_2} \ldots p_s^{k_s}$ е разлагане на числото *x* на прости множители, то броят на делителите на *x* е $(k_1+ k_2+...+ k_s)$. **Доказателство.** Действително, един прост делител *p* на *x* с кратност *k* може да участва като множител в делител на *x* точно *k* + 1 пъти – 0, 1, ..., *k* – 1 или *k*. Затова броят на делителите на *x* е произведението на увеличените с единица кратности на всички прости делители на *x*.

Лема 2. Ако x е цяло число от интервала $[1, t]$, което не е просто, тогава съществува негов прост делител p, за който е в сила $p^2 \le t$. **Доказателство.** Виж решението към задача D3.

Тъй като $32000^2 > 10^9$, отначало намираме в масива `int prime[4000]` всички прости числа по-малки от $m = 32000$, като използваме решето на Ератостен:

```
int sieve[32000]={0};
int prime[4000];
int pcnt;
void genprimes(int m)
{  pcnt = 0;
   for(int d=2;d<m;d++)
      if(sieve[d]==0)
      { pcnt++; prime[pcnt]=d;
        for(int j=2*d; j<m; j=j+d)
        sieve[j] = 1;
      }
}
```

Да означим полученото множество от прости числа с P. Оказва се, че множеството P съдържа точно 3432 числа, като най-голямото от тях е 31991, при това $31991^2 > 10^9$. Нека x е едно от дадените N числа. Намираме кратността на числата от множеството P в разлагането на x на прости множители и прилагайки Лема 1 пресмятаме броя на делителите.

```
int divcnt(int x)
{ int dcnt=1;
  for(int i=1; prime[i]*prime[i]<=x; i++)
  { int k=0;
    while(x%prime[i] == 0)
    { x = x/prime[i];
      k++;
    }
    dcnt = dcnt * (k+1);
  }
  if(x>1) // ако x е било просто число
    dcnt = dcnt*2;
  return dcnt;
}
```

Остава да добавим главната програма:

```
int main()
{  genprimes(32000);
   int n,x;
   cin >> n;
   for(int k = 1; k<n; k++)
   {  cin >> x;
      cout << divcnt(x) << " ";
   }
   cin >> x;
   cout << divcnt(x) << endl;
   return 0;
}
```

ТЕМА ЗА ГРУПА D (6 КЛАС)

Задача D1. ПРИЯТЕЛИ, автор Стоян Капралов

Три момчета трябвало да пренесат общо N кутии. Напишете програма **box**, която определя по колко различни начина момчетата могат да разпределят работата помежду си, ако всеки трябва да пренесе поне една кутия.

Вход. От един ред на стандартния вход се въвежда цялото число N.

Изход. Резултатът да се изведе на стандартния изход.

Ограничения. $3 \le N \le 100$.

ПРИМЕР

Вход:	**Изход:**
5	6

Обяснение на изхода. Възможни са 6 начина за разпределяне на работата: 1,1,3; 1,2,2; 1,3,1; 2,1,2; 2,2,1; 3,1,1.

РЕШЕНИЕ

Преди да започнем да броим възможните случаи, нека първо отделим по една кутия за всяко от момчетата, за да сме сигурни, че всеки е получил поне една кутия. Остава да видим по колко начина останалите $N - 3$ кутии могат да се разпределят между тримата. Нека първото момче не е взело нито една от останалите кутии. Тогава другите двама могат за да си разделят тези кутии по $N - 2$ начина: 0 за второто и $N - 3$ за третото, 1 за второто и $N - 4$ за третото, и т.н., $N - 3$ за второто и 0 за третото. Ако сега първото момче вземе 1 от останалите кутии, то оставащите $N - 4$ кутии могат да се разпределят по $N - 3$ начина. Разсъждавайки по този начин установяваме, че ако първото момче вземе $N - 2$ от кутиите, останалата една кутия може да се разпредели между другите двама по два начина. А ако първото момче вземе всичките $N - 3$ кутиите, не остава нищо за разпределяне между другите двама и това е едно възможно разпределяне. Следователно търсеният брой можем да намериме, като сумираме числата 1, 2, ..., $N - 2$.

```
#include <iostream>
using namespace std;
int main()
{  int n,i,br=0;
   cin>>n;
   for(i=1;i<=n-2;i++) br=br+i;
   cout<<br<<endl;
}
```

Но дори тази проста програма не е най-доброто решение на задачата. Как постъпил великият математик Карл Фридрих Гаус, когато бил на възраст да се състазава в група D и трябвало да реши подобна задача – да намери сумата на числата от 1 до *M*. Той събрал първото число с последното и получил сума $M + 1$, второто с предпоследното и получил същата сума, и така нататък. Следователно, разсъждавал малкият Гаус, всяко от числата, събрано със съответното му от другия край на последователността ще даде сума $M + 1$. Ако съберем всички тези суми, ще получим $M.(M + 1)$, а тъй като всяко число е събрано два пъти, търсената сума ще бъде $M.(M + 1)/2$.

```
#include <iostream>
using namespace std;
int main()
{  int n; cin>>n;
   cout<<(n-2)*(n-1)/2<<endl;
}
```

Задача D2. ЦИФРИ, автор Сюзан Феимова

Дадено е цяло положително число *N*, записано в десетична бройна система. За всяка от десетичните цифри, участващи в него, определяме тегло, което се изчислява като сбора от всички такива цифри, участващи в числото плюс техния брой. Например, в числото 553333, цифрата 5 има тегло 5 + 5 + 2 = 12, а цифрата 3 има тегло 3 + 3 + 3 + 3 + 4 = 16. Напишете програма **digits**, която размества цифрите на зададеното число така, че в полученото ново число еднаквите цифри да са записани една след друга, като по-наляво да се поставят цифрите с по-малко тегло. Ако има цифри с равно тегло, то по-напред да се запишат по-големите цифри.

Вход. На един ред на стандартния вход е зададено числото *N*.

Изход. На един ред на стандартния изход програмата трябва да изведе едно цяло число, съставено от цифрите на въведеното число *N*, подредени според изискванията на задачата.

Ограничения. $10^2 < N < 10^{40}$

ПРИМЕР

Вход:	**Изход:**
40201201015101040	52200000004411111

РЕШЕНИЕ

Числото *N* е в интервала $[10^2,10^{40}]$ и не се „вмества“ в целочислена променлива от стандартен тип. Затова да въведем числото като низ в масива `char s[41]`. Определяме броя `n` на цифрите в низа, чрез функцията `strlen()`.

Според условието на задачата трябва да сортираме цифрите на зададеното число според теглото им, където **теглото=цифрата* броя + броя = броя*(цифрата + 1**). За целта да поставим цифрите от 0 до 9 в масива `int c[10]`, а теглата им – в масива `int t[10]`. Цифрата `i` ще бъде в `c[i]`, а теглото ѝ – в `t[i]`. За да определим теглата на цифрите ще прегледаме масива `s` и за всяка срещната в него цифра `j=s[i]-'0'` ще увеличим `t[j]`, съгласно формулата за теглото, с `j+1`. Сега можем да сортираме във възходящ ред масива `t`, съдържащ теглата на цифрите, като правим необходимите разместванията и в масива `c` със самите цифри. Тъй като броят на цифрите не надвърля 10, не е нужен бърз алгоритъм за сортиране – в авторското решение е използван алгоритъмът с намиране на минимален елемент.

Остава да изведем подредените според теглата си цифри. Пропускаме отляво надясно всички цифри с нулеви тегла – очевидно е, че те не участват в *N*. Ако първата цифра с ненулево тегло е 0, пропускаме и нея, защото няма да извеждаме в резултата водещите нули. Всяка от останалите цифри извеждаме толкова пъти, колкото се срещат в *N*. Броят на срещанията, може да се изчисли от теглото на цифрата: **брой = тегло / (цифра + 1)**. Затова броят на цифрите `c[i]` в *N* е `t[i]/(c[i]+1)`.

```
#include<iostream>
using namespace std;
int t[10],c[10],i,j,k;
char s[41];
int main()
{
 cin>>s;
 int n=strlen(s);
 for(i=0;i<=9;i++) {c[i]=i;t[i]=0;}
 for(i=0;i<n;i++) {j=s[i]-'0';t[j]+=j+1;}
 for(j=0;j<9;j++)
    for(i=j+1;i<=9;i++)
       if((t[i]<t[j])||(t[i]==t[j] && c[i]>c[j]))
         { swap(t[i],t[j]); swap(c[i],c[j]); }
k=0; while(t[k]==0) k++;
if(c[k]==0) k++;
for(i=k;i<=9;i++)
  for(j=0;j<t[i]/(c[i]+1);j++) cout<<c[i];
 cout<<endl;
 return 0;
}
```

Задача D3. ШИШО

Таксиметровата фирма „Бам-Бам“ има свой собствен паркинг. На него могат да паркират безплатно коли на фирмата, но често се случва някои недобросъвестни шофьори да се възползват от паркинга. За да предпази фирмата от тези нахалници, шефът ѝ Шишо Бакшишо пререгистрирал всичките си таксита, като избрал новите им номера да бъдат прости числа. Инсталирал и система, която да заснема номера на колата и, в зависимост от това дали номерът е просто число или не, да вдига, или да не вдига бариерата. Една кола може да влиза в паркинга много пъти за разглеждания период от време. Но поради некадърността на програмистите му,

системата снимала номерата на колите в огледален образ. Ако снимката показва 0173, истинският номер на колата е 3710.

Като добри програмисти, на вас се разчита да помогнете на Шишо Бакшишо и да напишете програма **shisho**, която по зададена последователност от заснети номера на коли, да изведете броя на тези, които трябва да бъдат допуснати в паркинга.

Вход. На първия ред на стандартния вход е зададено цялото положително число *N* – броят на заснетите номера на коли. На втория ред на стандартния вход са зададени *N*-те номера на коли от снимките. Всяка снимка е сканирана до последователност от 4 цифри, от които поне една не е 0, а отделните четворки от цифри са разделени с по един интервал.

Изход. Програмата трябва да изведе на един ред на стандартния изход броя на колите, които са допуснати в паркинга.

Ограничения. $0 < N \leq 50000$, 0001 ≤ номер на кола ≤ 9999.

ПРИМЕР 1	**ПРИМЕР 2**
Вход:	**Вход:**
4 3000 0013 8009 7100	6 0870 4412 0981 4989 4142
Изход:	**Изход:**
2	0

РЕШЕНИЕ

Нека в променливата `int n` сме въвели броя на снимките. В променливата `int x` ще въвеждаме номерата на колите, така както са сканирани и в същата променлива ще пресмятаме правилният номер (съставен от същите цифри, но в обратен ред). С променливата `int res = 0` ще броим на колите които изпълняват поставеното условие – да са прости числа.

Подпрограмата `int rev(int n)` получава като аргумент цяло положително четирицифрено число `n` и връща като резултат числото съставено от същите цифри, но в обратен ред:

```
int rev(int n)
{
   int a=n%10; n=n/10;
   int b=n%10; n=n/10;
   int c=n%10; n=n/10;
   int d=n;
   return a*1000+b*100+c*10+d;
}
```

Подпрограмата `bool prime(int n)` проверява дали зададеното като аргумент положително число е просто,като преброява числата които го делят (числото не е просто ако броят на делителите му е по-голям от 2):

```
bool prime(int n)
{
    if (n == 1) return false;
    int d = 0;
    for (int i = 1; i <= n; i++)
        if (n % i == 0) d++;
    return d > 2;
}
```

За да завършим решението остава да добавим към тези две подпрограми главната функция:

```
#include <iostream>
using namespace std;
int rev(int);
bool prime(int);
int main()
{  int n,x,res=0;
   cin >> n;
   for(int i=0;i<n;i++) {
      cin>>x; x=rev(x);
      if (prime(x)) res++;
   }
   cout<<res<<endl;
   return 0;
}
```

В подпрограмата `bool` проверката за простота на числото `n` изисква `n` деления. Предлагаме следното подобрение – за да установим дали едно число `n` е просто, трябва да проверим дали не се дели само на прости числа. При това няма нужда да проверяваме всички прости числа `p` по-малки от `n`, а само такива, че `p*p<=n`. Действително, ако `n` има прост делител `p`, `p*p>n`, то `n=pq` и `n` има делител `q`, `q*q<n`. Ако `q` е просто, вече сме намерили този прост делител на `n`, преди да стигнем до `p`. А ако `q` не е просто, то `q` има по-малък прост делител `q'` и вече сме намерили този делител на `n`, преди да стигнем до `q`. За да реализираме алгоритъма е необходимо да намерим всички прости числа `p`, `p*p<=10000`, т.е. `p<=100`. Това можем да направим на ръка или като си напишем не сложна помощна програма. Поставяме получените 25 прости числа в масива `int p[25]={2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97};`

и променяме подпрограмата `prime`, както следва:

```
bool prime(int n)
{
   int d=1;
   for(int i=0;i<=24&&p[i]*p[i]<=n;i++)
      if(n%p[i]==0) {d=0; break;}
   return d;
}
```

ТЕМА ЗА ГРУПА Е (4–5 КЛАС)

Задача Е1. ПОТРЕБИТЕЛСКО ИМЕ (НИКНЕЙМ), автор Явор Никифоров

Ето, че и сегашният никнейм: 5l<%P3_FFFE\\G@+/\–lIgLic4hkayyy!!1! на скайп-фенката Лигличкатъ взе да й омръзва. Не може и да я вините – само от 15 минути е в Skype-а, а някои от приятелите й вече я разпознаха (което всъщност трябваше да се очаква). Затова е нетърпелива да последва днешната мода – да си избере за никнейм някаква дълбока мисъл, изписана с редуващи се главни и малки букви: `НеЩо-тАкОвА-СеЩаШ-Ли-сЕ-КоЛкО-Е-ЯкО`. А това е изключително затормозяващо! Трябва постоянно да внимаваш и да помниш четността на поредния номер на текущата буква заради правилото:

- ако на нечетна позиция в ника има буква – тя непременно трябва да е главна;
- ако на четна позиция в ника има буква – тя непременно трябва да е малка;
- цифрите и другите знаци, които не са букви – не се променят, независимо на каква позиция са;
- интервали и букви на кирилица в ника няма да има, нито пък точки, с изключение на една за край на цялото това велико умотворение.

Напишете програма **`nick`**, която получава като вход мъдрата мисъл, която нашата героиня си е избрала за скайп-име и извежда същия текст, променен до неузнаваемост, както повелява скайп-модата. Позициите в текста броим от 1, започвайки от най-левия знак.

Вход. От един ред на стандартния вход се въвежда текст без интервали, без букви на кирилица и със само един знак точка в края.

Изход. На един ред на стандартния изход програмата трябва да изведе видоизменения по горните правила текст.

Ограничения. Дължината на текста няма да е по-голяма от 1000.

ПРИМЕР

Вход:
`Moje_da_si_GROZ3N,no_za_smetka_na_t'va_si_TUP!!!1!!11!.`
Изход:
`MoJe_dA_Si_gRoZ3N,No_zA_SmEtKa_nA_T'Va_sI_TuP!!!1!!11!.`

РЕШЕНИЕ

За решаването на задачата е необходима една променлива `c` от тип `char`, в която да въвеждаме буквите на текста и една променлива `br` от тип `int`, с която да следим поредния номер на буквата в текста. В началото на промеливата `br` даваме стойност 0. Най-лесно е да обработим текста с цикъл `do-while`, като го прекратим, след като сме обработили точката, определяща края на текста. За всеки въведен знак увеличаваме с единица брояча `br`, за да получим поредния номер а този знак в текста. Ако поредният номер е нечетно число и знакът е малка латинска буква – зменяме тази буква със съответната голяма (намаляваме ASCII кода й с разликата `'a'-'A'`). Ако поредният номер е четно число и знакът е голяма латинска буква – зменяме тази буква със съответната малка (увеличаваме ASCII кода й с разликата `'a'-'A'`). Останалите знаци остават непроменени. Преди да направим седваща стъка на цикъла, извеждаме получения в резулат от обработката знак.

```
#include <iostream>
using namespace std;
int main()
{
   char c;int br=0;
   do {
      cin>>c; br++;
      if(br%2==1&&c>='a'&&c<='z') c=c-('a'-'A');
      if(br%2==0&&c>='A'&&c<='Z') c=c+('a'-'A');
      cout<<c;
   } while (c!='.');
   return 0;
}
```

Задача Е2. СИМПАТИЧНА РЕДИЦА, автор Бистра Танева

Една редица от цели числа наричаме *симпатична*, ако има поне две съседни числа, които са рвни по стойност. Напишете програма **`lovable`**, която проверява дали дадена редица е симпатична.

Вход. От първия ред на стандартния вход се въвежда броят N на числата в редицата. От втория ред на стандартния вход се въвеждат N цели числа, разделени с по един интервал, представляващи елементите на редицата.

Изход. Ако редицата е симпатична, програмата трябва да изведе на стандартния изход числото от най-дългата последователност равни числа, която се среща в редицата. Ако има няколко еднакво дълги последователности, програмата трябва да изведе число от тази последователност, която се среща най-рано в редицата. Ако редицата не е симпатична, програмата трябва да изведе на стандартния изход само `no`.

Ограничения. $2 \le N \le 100$. Всяко от числата е положително и не по-голямо от 1 000 000 000 000.

ПРИМЕР 1

Вход:
4 3 7 2 3
Изход:
No

ПРИМЕР 2

Вход:
9 15 2 2 2 4 4 4 19 2
Изход:
2

РЕШЕНИЕ

Да разделим решението на задачата на две отделни части – намиране на **дължината на последователност от еднакви числа** и намиране на **максималната дължина**

на такава последователност. В променливата k ще съхраняваме числото, което е възможно начало на нова последователност от еднакви числа, а в променливата ch – последното прочетено число (заради възможността числата в редицата да са много големи, тези променливи трябва да са от тип **long long**). Броят на еднаквите с k числа ще намерим в променливата br. Въвеждаме дължината на редицата в променливата n, първото число в променливата k, а в брояча br поставяме 1. С цикъла for обработваме останалите числа на редицата – от второто до последното. Въвеждаме поредното число в ch и ако то е равно на числото в k – увеличаваме брояча с 1 и въвеждаме следващото число.

Когато числото в ch не е равно на числото в k – завършила е последователност от еднакви числа и трябва да проверим дали нейната дължина, намираща се в br, не е по-голяма от намерената до момента. За целта използваме променливата maxbr, в която преди началото на цикъла сме поставили 1. Ако br>maxbr, намерена е по-дълга последователност от запомнената до момента. Тогава заменяме стойността в maxbr с br, а в променливата maxch запомняме числото от k, което образува тази редица (не от ch, защото там е началото на новата подредица!). Независимо от това дали сме намерили по-дълга редица или не, връщаме брояча в начално положение 1, заменяме стойността на k с тази на ch и продължаваме изпълнението на цикъла. Когато редицата завършва с последователност от еднакви числа, тогава нейната дължина не е проверена и проверката за максимална дължина трябва да се повтори още един път след края на цикъла.

Ако съдържанието на maxbr е останало 1, в редицата няма последователност от еднакви числа. Тогава на стандартния изход извеждаме низа "no". Ако съдържанието на maxbr е по-голямо от 1, в редицата има последователност от еднакви числа и тогава на стандартния изход извеждаме стойността на maxch. Задачата изисква от всички най-дълги последователности да се изведе тази, която е най-близо до началото на редицата. Ако трябваше да намерим най-близката до края на редицата последователност, тогава сравнението br>maxbr трябва да стане br>=maxbr.

```
#include <iostream>
using namespace std;
int main()
{
   long long k, ch, maxch;
   int n, br, maxbr=1;
   cin>>n; cin>>k; br=1;
   for(int i=2;i<=n;i++) {
      cin>>ch;
      if(ch==k) br++;
      else {
         if (br>maxbr) {maxbr=br;maxch=k;}
           k=ch;br=1;
      }
   }
   if (br>maxbr) {maxbr=br;maxch=k;}
   if(maxbr==1) cout<<"no"<<endl;
   else cout<<maxch<<endl;
   return 0;
}
```

Задача Е3. СТАДИОН, автор Бисерка Йовчева

Ангел, Боби и Цецо обичат да посещават училищния стадион, където могат да тренират различни спортове. Един ден тримата приятели започнали да спорят, кой от тях е тренирал най-упорито. Добре, че всеки ден учителят по физкултура записвал часа и минутата на пристигане и тръгване на всеки ученик. За да разрешат спора, те помолили учителя по физическо за неговите записки.

Така приятелите получили три редици от по четири числа. Първата редица съдържала часа и минутите на пристигане, както и часа и минутите на тръгване на Ангел. Във втората четворка били зададени часът и минутите на пристигане и тръгване на Боби, а в третата – часът и минутите на пристигане и тръгване на Цецо. Нашите приятели били отлични спортисти, но много лоши математици. Налага се да им помогнете в спора, като напишете програма **stadion**, която определя кой от тях, колко е упорит в тренировките.

Вход. На три последователни реда на стандартния вход ще бъдат зададени, по начина, описан по-горе, по четири цели числа – времената (час и минути) на пристигане и заминаване на всеки от приятелите.

Изход. Програмата трябва да изведе на стандартния изход низ, съставен от главните латински букви A (за Ангел), B (за Боби), C (за Цецо) и X. Последователността на буквите A, B и C в низа, трябва да съответства на продължителността на тренировката на съответния ученик – колкото повече е тренирал един ученик, толкова по-наляво в низа е буквата му. Ако няколко от приятелите са тренирали еднакво дълго – техните букви се заменят от една буква X.

Ограничения. Всички зададени във входа времена са в рамките на едно денонощие. Времето на пристигане на всеки от приятелите е по-ранно от времето му на тръгване.

ПРИМЕР 1

Вход:
```
11 5 11 59
11 30 12 7
12 10 13 3
```
Изход:
```
ACB
```

ПРИМЕР 2

Вход:
```
11 5 11 59
11 30 12 7
12 10 13 4
```
Изход:
```
XB
```

РЕШЕНИЕ

Тъй като всички зададени времена в задачата са в рамките на едно денонощие, лесно е да се намерят продължителностите на пребиваване на всеки от приятелите на стадиона – изразени в минути. Нека след въвеждане на времената на пристигане и тръгване на всеки от тримата ученици сме пресметнали тези продължителности в променливите `a`, `b` и `c`, съответно. По-лесният, но по-дълъг за изписване и правещ повече проверки алгоритъм, проверява за всеки от възможните случаи преди да изведе съответно съобщение. А възможните случаи са 13 – един случай, когато трите времена са равни, 6 случая когато две от времената са равни, а третото е различно и 6 случая, когато и трите времена са различни.

```
#include<iostream>
using namespace std;
int main()
{
    int h,m,a,b,c;
    cin>>h>>m; a=60*h+m;
    cin>>h>>m; a=60*h+m-a;
    cin>>h>>m; b=60*h+m;
    cin>>h>>m; b=60*h+m-b;
    cin>>h>>m; c=60*h+m;
    cin>>h>>m; c=60*h+m-c;
    if(a==b&&a==c) cout<<"X"<<endl;
    if(a==b&&a>c) cout<<"XC"<<endl;
    if(a==b&&a<c) cout<<"CX"<<endl;
    if(a==c&&a>b) cout<<"XB"<<endl;
    if(a==c&&a<b) cout<<"BX"<<endl;
    if(b==c&&b>a) cout<<"XA"<<endl;
    if(b==c&&b<a) cout<<"AX"<<endl;
    if(a>b&&b>c) cout<<"ABC"<<endl;
    if(a>c&&c>b) cout<<"ACB"<<endl;
    if(b>a&&a>c) cout<<"BAC"<<endl;
    if(b>c&&c>a) cout<<"BCA"<<endl;
    if(c>a&&a>b) cout<<"CAB"<<endl;
    if(c>b&&b>a) cout<<"CBA"<<endl;
    return 0;
}
```

Възможен е и друг алгоритъм. Да подредим получените три числа по големина – най-голямото в `a`, средното в `b` и най-малкото в `c`. За да можем да изведем искания низ, трябва да поредим и трите букви, с които сме означили учениците, съответен на подреждането на времената. Нека буквите са в променливите (от тип `char`) `la`, `lb` и `lc`, съответно. За подреждане на трите времена използваме следната последователност от операции: ако първото число е по-малко от второто – разместваме тези две числа; ако това което се окаже на второ място е по-малко от третото – разместваме двете числа; ако сега първото число се окаже по-малко от второто – отново разместваме тези две числа. Винаги когато разместваме две числа, разместваме и съответните им букви. След като числата са подредени в намаляващ ред, броят на случаите, които трябва да разгледаме за да определим низа, който да изведем, намалява от 13 на 4, но в замяна на това са необходими много операции за подреждането на числата. Ето фрагмента, реализиращ тази идея:

```
if(a<b){t=a;a=b;b=t;lt=la;la=lb;lb=lt;}
if(b<c){t=b;b=c;c=t;lt=lb;lb=lc;lc=lt;}
if(a<b){t=a;a=b;b=t;lt=la;la=lb;lb=lt;}
if(a==b&&a==c) cout<<"X"<<endl;
if(a==b&&a>c) cout<<"X"<<lc<<endl;
if(b==c&&a>b) cout<<la<<"X"<<endl;
if(a>b&&a>c) cout<<la<<lb<<lc<<endl;
```

Редактор на брошурата: Кр. Манев