Міністерство освіти і науки України
Львівський фізико-математичний ліцей
при Львівському національному університеті
імені Івана Франка

Міжнародний конкурс з інформатики та комп'ютерної вправності «Бебрас–2017»

Матеріали школи програмування переможців конкурсу (Львів–2017)

Кам'янець-Подільський
«Аксіома»
2018

УДК 519.683
М58

Упорядник:
Ростислав Шпакович

М58 Міжнародний конкурс з інформатики та комп'ютерної вправності «Бебрас–2017». Матеріали школи програмування переможців конкурсу (Львів–2017) / С. С. Жуковський, І. М. Порубльов, І. В. Скляр, Р. С. Шпакович – Кам'янець-Подільський : Аксіома, 2018. – 96 с.

ISBN 978-966-496-448-4

У посібнику подано теоретичний матеріал і задачі, які опрацьовувались слухачами Школи програмування переможців конкурсу «Бебрас», що проходила у жовтні 2017 року в м. Львові. Ці матеріали будуть корисними вчителям інформатики й учням під час вивчення основ програмування та підготовки до олімпіад.

Видання здійснено на благодійницьких засадах і розповсюджується безкоштовно

УДК 519.683

ISBN 978-966-496-448-4

Передмова

Усе більш масовим у світі стає Міжнародний конкурс з інформатики та комп'ютерного мислення «Бебрас» (у перекладі з литовської – «Бобер»). Міжнародний оргкомітет вибрав таку офіційну назву, визнаючи заслуги інформатиків Литви у створенні та популяризації конкурсу в усьому світі.

У конкурсі «Бебрас–2017» узяли участь понад 2 мільйони 160 тисяч учнів із 44-х країн світу. В Україні конкурс пройшов удесяте. Детальніше – у статті «Конкурс «Бебрас–2017» у світі та Україні».

Уже чотири роки у м. Львові проводиться Осіння школа програмування для переможців конкурсу «Бебрас». Остання школа відбулась 13–19 жовтня 2017 року.

Цього разу в ній узяли участь 27 учнів із 9 областей України (Дніпропетровської, Закарпатської, Івано-Франківської, м. Києва, Кіровоградської, Львівської, Полтавської, Сумської, Чернігівської).

Упродовж 7 днів Школи учасники навчалися й удосконалювали свої знання з програмування. Заняття проводили досвідчені викладачі, автори відомих книг із програмування та задач Всеукраїнських олімпіад і турнірів:
Сергій Жуковський, доцент Житомирського державного університету;
Ілля Порубльов, старший викладач Черкаського національного університету;
Ірина Скляр, викладач Київського природничо-наукового ліцею.

Заняття проводились паралельно у двох групах:

1-ша група – переможці обласних і Всеукраїнських олімпіад.

Теми занять:

1-й день – Ілля Порубльов. Обчислювальна геометрія.

2-й день – Ілля Порубльов. Злиття та два вказівники.

3-й день – Сергій Жуковський. Дерево Фенвіка та дерево відрізків.

2-га група – учні, які роблять перші кроки в олімпіадному програмуванні.

Теми занять:

1-й день – Ірина Скляр. Теорія чисел.

2-й день – Сергій Жуковський. Бінарні операції.

3-й день – Ірина Скляр. Динамічне програмування.

В останній день була проведена олімпіада.

Її переможцями стали:

11-й клас

1. Нікіта Пупов (СЗШФМП № 12, м. Чернігів).
2. Лев Потьомкін (ПНЛ № 145, м. Київ).
3. Станіслав Домбровський (СШ № 5, м. Івано-Франківськ).

10-й клас

1. Борис Нижник (ПНЛ № 145, м. Київ).
2. Андрій Затилюк (УФМЛ, м. Київ).
3. Дмитро Столинець (СЗШФМП № 12, м. Чернігів).

9-й клас

1. Дар'я Нємкевич (СЗШФМП № 12, м. Чернігів).
2. Владислав Москаленко (Олександрійський ЛІТ, Кіровоградська обл.).
3. Андрій Семенистий (Шосткинський НВК, Сумська обл.).

Вісім учнів цьогорічної Школи стали учасниками 4-го туру Всеукраїнської олімпіади з інформатики 2018 року, п'ятеро з них – Борис Нижник, Лев Потьомкін, Нікіта Пупов, Дар'я Нємкевич і Владислав Москаленко стали призерами.

У статтях авторів цієї книжки подано теоретичний матеріал і задачі, які опрацьовувались слухачами Школи. Ці матеріали будуть корисними вчителям інформатики й учням під час вивчення основ програмування та підготовки до олімпіад.

Сергій Жуковський, кандидат педагогічних наук, доцент кафедри прикладної математики та інформатики Житомирського державного університету імені Івана Франка.

Дерево Фенвіка. Дерево відрізків

Обчислення суми елементів масиву на певному інтервалі є популярною задачею. Досить часто потрібно проводити статистичну обробку баз даних за певні періоди. Наприклад, знаходити критичні значення на інтервалах (максимальні, мінімальні значення курсу валют, цін на певні продукти, сумарні прибутки товарів за певні періоди, тощо). Такі запити часто задають користувачі різних сайтів.

Розглянемо одну задачу такого типу. Є статичний (незмінний) масив чисел, і багато запитів на підрахунок суми елементів цього масиву на деяких інтервалах від L до R.

Повний перебір усіх елементів масиву доречно робити тільки один раз, коли розмір масиву і кількість запитів у межах до 106, алгоритм повного перебору всіх елементів для кожного запиту працюватиме повільно, кількість операцій становитиме 1012 операцій.

Нехай задано такий масив:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	3	1	5	4	5	3	2	1	2	3	4	3	2	4	3	4	3	4	1

Для прискорення виконання цієї задачі створимо масив часткових сум:

i-тий елемент цього масиву – сума всіх елементів заданого масиву від першого до i-того включно. Це можна реалізувати за n операцій.

```
for(i=0;i<=n;i++)
    b[i]=b[i-1]+a[i];
```

В утвореній таблиці кожен запит можна виконати за одну операцію.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	5	6	11	15	20	23	25	26	28	31	35	38	40	44	47	51	54	58	59

$S = b[R] - b[L - 1]$.

Ускладнимо задачу.

Додамо запити на зміну елементів масиву. Будемо замінювати елементи масиву з номером t на значення v. Якщо запити зміни елементів та обчислення суми на інтервалі проходитимуть упереміжку, то використовуючи цей метод, доведеться постійно робити перерахунок масиву часткових сум.

Для швидкої реалізації перерахунку сум можна використати дві структури даних: дерево Фенвіка та дерево відрізків.

Дерево Фенвіка

Для реалізації цього методу Фенвік запропонував дві побітові функції.

$X \mid (X + 1)$ – побітове АБО

та $X \,\&\, (X + 1) - 1$

Наприклад, розглянемо функцію $FD(X) = X \mid (X + 1)$. Застосуємо її для деяких чисел $FD(1) = 3$; $FD(3) = 7$; $FD(7) = 15$; (ці числа можна представити у вигляді $2^n - 1$).

$FD(4) = 5$; $FD(5) = 7$; $FD(6) = 7$; $FD(8) = 9$; $FD(9) = 11$; $FD(10) = 11$; $FD(11) = 15$;

Для кращого розуміння обчислень наводимо таблицю двійкового представлення десяткових чисел:

0	00000
1	00001
2	00010
3	00011
4	00100
5	00101
6	00110
7	00111
8	01000
9	01001
10	01010
11	01011
12	01100
13	01101
14	01110
15	01111
16	10000
17	10001
18	10010
19	10011
20	10100

Можна помітити, що значення функції – це число, у якому останній нуль аргумента в двійковому записі замінюється на 1. Якщо виконати цю функцію над будь-яким числом, то за декілька операцій утворимо число, яке можна представити у вигляді $\mathbf{2^n - 1}$.

Будуємо дерево Фенвіка (новий масив) таким чином. Спочатку цей масив порожній. Далі послідовно кожний елемент $A[i]$ початкового масиву додаємо до елементів дерева Фенвіка з номерами i, $FD(i)$, $FD(FD(i))$ і т. д. Тобто, номер кожної наступної комірки рекурсивно визначається функцією FD.

Наприклад, на першій ітерації в усі комірки з номерами, які можна представити у вигляді $2^n - 1$, додаємо перше число.

На другій ітерації елемент $A[1] = 3$ додаємо у комірки з номерами 1, 3, 7, 15. На третій ітерації елемент $A[2] = 1$ додаємо у комірки з номерами 2, 3, 7, 15.

Після заповнення масиву таким чином у кожній комірці з номером $2^n - 1$ знаходитиметься сума всіх елементів масиву від початку до даного елемента включно:

№	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	2	3	1	5	4	5	3	2	1	2	3	4	3	2	4	3
1	2	2		2				2								2
2		5		5				5								5
3			1	6				6								6
4				11				11								11
5					4	4		15								15
6						9		20								20
7							3	23								23
8								25								25
9									1	1		1				26
10										3		3				28
11											3	6				31
12												10				35
13													3	3		38
14														5		40
15															4	44
16																47
t	2	5	1	11	4	9	3	25	1	3	3	10	3	5	4	47

```
void modif(int i, int d)
{
    while (i<n)
    {
            t[i]+=d;
            i=i|(i+1);
    }
}
```

Наступне завдання – швидко знайти суму елементів від будь-якого елемента до найближчого меншого елемента з номером $2^n - 1$.

Розглянемо другу функцію Фенвіка. $FU(x) = X \,\&\, (X + 1) - 1$.

Функція $FU(x)$ від будь-якого x, яке можна представити у вигляді $2^n - 1$, дорівнює –1.

$FU(12) = 11$; $FU(11) = 7$; $FU(10) = 9$; $FU(9) = 7$; $FU(8) = 7$; $FU(6) = 5$; $FU(5) = 3$; $FU(2) = 1$.

Коли просумуємо всі числа масиву t від даного індексу, пройшовши по елементах функції $FU(x)$, то отримаємо суму всіх елементів масиву A від початку до даного елемента.

```
int suma(int r)
{
    int result =0;
    while(r>=0)
    {
            result+=t[r];
            r=(r&(r+1))-1;
    }
    return result;
}
```

Для пошуку суми елементів масиву на проміжку від L до R потрібно знайти різницю $sum(R) - sum(L - 1)$.

Складність роботи алгоритму побудови дерева Фенвіка $O(N * log(N))$.

Складність заміни елемента масиву (модифікації дерева Фенвіка) $O(log(N))$.

Складність пошуку суми елементів масиву з використанням дерева Фенвіка 2*O(log(N)).

Розбір задач

Побудова дерева Фенвіка

https://www.e-olymp.com/uk/problems/8247

Задано масив A з n натуральних чисел. Побудуйте дерево Фенвіка і виведіть масив після кожної ітерації додавання елемента до дерева.

Вхідні дані

У першому рядку знаходиться розмір масиву n ($1 \le n \le 100$). У наступному рядку знаходяться n натуральних чисел – елементи масиву A ($1 \le A_i \le 10^9$).

Вихідні дані

Виведіть n рядків, у кожному з яких виведіть n чисел – масив після чергової ітерації додавання елемента до дерева Фенвіка.

Вхідні дані	Вихідні дані
6 5 4 4 2 3 1	5 5 0 5 0 0 5 9 0 9 0 0 5 9 4 13 0 0 5 9 4 15 0 0 5 9 4 15 3 3 5 9 4 15 3 4

Використаємо функцію **modif** для побудови дерева Фенвіка. Під час зчитування кожного елемента масиву викликатимемо функцію **modif**, і після кожного виклику функції **modif** виводитимемо масив, у якому зберігається дерево Фенвіка.

```
#include<bits/stdc++.h>
#define ll longlong
using namespace std;
ll a[105];
ll t[105]={0};
ll n=20;
void modif(ll i,ll d)
{
     while (i<n)
     {
              t[i]+=d;
              i=i|(i+1);
     }
}
void out(ll *mas,ll n)
{
     for(ll i=0;i<n;i++)
     cout<<mas[i]<<» «;
     cout<<endl;
}

int main()
{
     ll q,i;
     cin>>n;
     for(i=0;i<n;i++)
     cin>>a[i];
     for(ll i=0;i<n;i++)
     {
```

```
            modif(i,a[i]);
            out(t,n);
    }
return0;
}
```

Дмитрик і масив

https://www.e-olymp.com/uk/problems/2941

Мама подарувала Дмитрику масив довжиною n. Масив цей не простий, а особливий. Дмитрик може вибрати два числа i та d ($1 \le i \le n$, $1000 \le d \le 1000$) й елемент з індексом i магічно стає рівним d. Дмитрик бавиться зі своїм масивом, а мама час від часу ставить йому запитання — яка сума всіх чисел масиву з індексами від f до t? Дмитрик легко справився з цими запитаннями, а чи зможете ви?

Вхідні дані

У першому рядку знаходиться два цілих числа n та q ($1 \le n \le 5 * 10^5$, $1 \le q \le 10^5$) – кількість елементів у масиві й сумарна кількість операцій та запитів відповідно. У наступному рядку задано n чисел a_1, a_2, ..., a_n ($-1000 \le a_i \le 1000$) – початковий стан масиву. У наступних q рядках задані операції та запити. Перший символ у рядку може бути = або ?. Якщо рядок починається із символа «=», то це операція присвоювання. Далі йдуть значення i та d. Якщо рядок починається із символа «?», то це запит. Далі йдуть числа f і t ($1 \le f, t \le n$).

Вихідні дані

Для кожного запиту виведіть суму чисел у масиві з індексами від f до t, по одному результату в рядку.

Вхідні дані	Вихідні дані
3 3 1 2 3 ? 1 3 = 3 2 ? 1 3	6 5

```
#include<bits/stdc++.h>
using namespace std;
int a[500004];
int t[500005]={0};
int n=20;
```

```
int suma(int r)
{
     int result =0;
     while(r>=0)
     {
              result+=t[r];
              r=(r&(r+1))-1;
     }
     return result ;
}
void modif(int i,int d)
{
     while (i<n)
     {
              t[i]+=d;
              i=i|(i+1);
     }
}
void set_f(intind, intval)
{
     int d = val-a[ind];
     a[ind] = val;
     modif(ind,d);
}

int main()
{
     int q,i;
     cin>>n>>q;
     for(i=0;i<n;i++)
     cin>>a[i];
     for(int i=0;i<n;i++)
     modif(i,a[i]);
     int u,v;
     char c;
     for(i=0;i<q;i++)
     {
              cin>>c>>u>>v;
              if(c=='?')
              cout<<(suma(v-1)-suma(u-2))<<endl;
              else
              {
                set_f(u-1,v);
              }
     }
     return 0;
}
```

Дерево відрізків

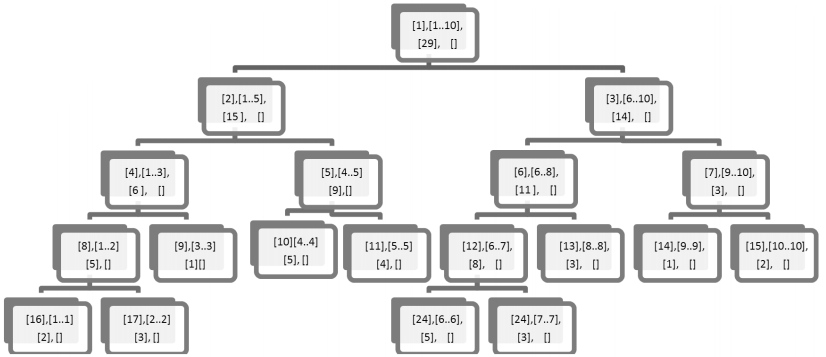
Дерево відрізків – це структура даних, що містить дані про масив і дає можливість швидко (за *log*) змінювати елемент масиву та швидко (за *log*) знаходити суму, максимальний, мінімальний елементи масиву.

Для реалізації дерева відрізків використовується масив розміром $4*N$, де N – розмір масиву, над яким виконуються операції.

Принцип побудови дерева відрізків такий. В першу комірку дерева помістимо суму всього масиву з інтервалу від 1 до N, в другу і третю відповідно суму першої половини масиву (від 1 до $N/2$) другої половини (від $N/2 + 1$ до N) і т. д.

Побудова дерева відрізків відбувається рекурсивно.

1	2	3	4	5	6	7	8	9	10
2	3	1	5	4	5	3	2	1	2




```
void build(int v,int TL,int TR)
{
    if(TL==TR) T[v]=a[TL];
    else
    {
            int m=(TL+TR)/2;
            build(2*v,TL,m);
            build(2*v+1,m+1,TR);
            T[v]=T[2*v]+T[2*v+1];
    }
}
```

Масив a – вхідний масив, над яким виконуються операції зміни масиву та пошук відповідно суми, мінімуму та максимуму.

Масив T – дерево відрізків.

V – вершина дерева відрізків, якій відповідає відрізок $[TL,\ TR]$.

Із кожної вершини відрізок поділяється на дві половини.

```
int m=(TL+TR)/2;
```

І рекурсивно заходимо у вершини з номерами 2*v та 2*v+1;

```
build(2*v,TL,m);
build(2*v+1,m+1,TR);
```

Коли довжина відрізка дорівнює 1, тоді додаємо у відповідну вершину дерева значення елемента масиву.

Після того як опрацюємо ліву та праву гілки з даної вершини, у дану вершину занесемо суму (максимум, мінімум) лівої та правої гілок відповідно.

Побудова дерева відбувається за $O(N)$ операцій.

Зміна елемента масиву потребує оновлення дерева. Для цього потрібно спуститися до листка дерева, змінити цю вершину (листок) і в зворотному порядку оновити дерево.

Функція оновлення дерева відрізків:

```
void update(int v,int TL,int TR, intnum, intval)
{
     if(TL==TR) T[v]= val;
     else
     {
              int m = (TL+TR)/2;
              if (num <= m) update(2*v,TL,m,num,val);
              elseupdate(2*v+1,m+1,TR,num,val);
              T[v] = T[2*v] + T[2*v+1];
     }

}
```

`num` – номер елемента масиву, який потрібно змінити;

`val` – нове значення елемента масиву.

Для пошуку суми (максимуму, мінімуму) використовують функцію:

```
int sum(int v,int TL,int TR,int l,int r)
{
     if(l>r) return0;   //(1)
     if(TL==l&&TR==r) return T[v];
     else
     {
              int m=(TL+TR)/2;
              returnsum(2*v,TL,m,l,min(m,r)) +
sum( 2 * v + 1, m + 1 , TR, max( m+1, l), r);     //(2)
     }
}
```

Для пошуку мінімуму (1) фунція повертає ∞, для максимуму $-\infty$, а в (2) відповідно мінімум і максимум відповідних елементів.

Задачі

Розв'язок задачі «**Дмитрик і масив**» із застосуванням дерева відрізків.

```
#include<bits/stdc++.h>
using namespace std;
int a[500006], T[2000005];
void build(int v,int TL,int TR)
{
     if(TL==TR) T[v]=a[TL];
     else
     {
              int m=(TL+TR)/2;
              build(2*v,TL,m);
              build(2*v+1,m+1,TR);
              T[v]=T[2*v]+T[2*v+1];
     }
}

int sum(int v,int TL,int TR,int l,int r)
{
     if(l>r) return 0;
     if(TL==l&&TR==r) return T[v];
     else
     {
              int m=(TL+TR)/2;
              return sum(2*v,TL,m,l,min(m,r)) + sum( 2 * v + 1, m + 1 ,
                       TR, max( m+1, l), r);
     }
}
```

```
void update(int v,int TL,int TR, intnum, intval)
{
     if(TL==TR) T[v]= val;
     else
     {
              int m = (TL+TR)/2;
              if ( num<= m) update(2*v,TL,m,num,val);
              else update(2*v+1,m+1,TR,num,val);
              T[v] = T[2*v] + T[2*v+1];
     }

}

int main()
{
     freopen(«input.txt»,»r»,stdin);
     freopen(«output.txt»,»w»,stdout);
     int n,i,k;
     cin>>n>>k;
     for(i=1;i<=n;i++)
     cin>>a[i];
     build(1,1,n);
     for(i=0;i<k;i++)
     {
              char t;
              int u,v;
              cin>>t>>u>>v;
              if(t==’=’) update(1,1,n,u,v);
              elsecout<<sum(1,1,n,u,v)<<endl;
     }
     return 0;
}
```

Пригоди Незнайка та його друзів

Усі ми пам’ятаємо історію про те, як Незнайко зі своїми друзями подорожували на повітряній кулі. Але не всі чоловічки помістились у кулі, тому що у неї була обмежена вантажопідйомність.

У цій задачі вам необхідно визначити, скільки чоловічків полетіло подорожувати. Відомо, що посадка у кулю не є оптимальною, а саме: чоловічки сідають у кулю у тій черзі, у якій вони стоять, як тільки комусь із них не вистачає місця, він і усі чоловічки, що залишились у черзі, повертаються і йдуть додому.

Вхідні дані

У першому рядку міститься кількість чоловічків n $(\mathbf{1 \le n \le 10^6})$ у Квітковому місті. У другому рядку вага кожного з чоловічків

у тому порядку, у якому вони сідатимуть у кулю. Усі ваги – натуральні числа і не перевищують $\mathbf{10^9}$. Далі йде кількість запитів m $\mathbf{(1 \le m \le 10^5)}$. Кожен запит являє собою один рядок. Якщо перше число у рядку дорівнює одиниці, то далі йде ще одне число v $\mathbf{(1 \le v \le 10^9)}$ – вантажопідйомність повітряної кулі. Якщо ж рядок починається з двійки, то далі йде два числа x $\mathbf{(1 \le x \le \underline{n})}$ та y $\mathbf{(1 \le y \le 10^9)}$ – це означає, що вага чоловічка, який стоїть на позиції $\boldsymbol{x}$, тепер дорівнює $\boldsymbol{y}$.

Вихідні дані

Для кожного запиту, що починається з одинички, виведіть в окремому рядку кількість чоловічків, які помістились у кулю.

Вхідні дані	Вихідні дані
5 1 2 3 4 5 5 1 7 1 3 2 1 5 1 7 1 3	3 2 2

Розв'язання

Побудуємо дерево відрізків для вхідного масиву. Для зміни масиву використаємо оновлення масиву, а для пошуку кількості чоловічків, які помістяться в кулю, опускатимемося по дереву в найлівішу вершину кожного рядка, у якій сума більша за допустиму. Вершина, у якій зупинимося, це номер першого учасника, який не поміститься в кулю.

```
#include<bits/stdc++.h>
#include<vector>
using namespace std;
long long a[1000001], o, t[4000004]={0};

void build (long long v, long long tl, long long tr)
     {
if (tl==tr) t[v]=a[tl];
else {
int m=(tl+tr)/2;
build (2*v, tl, m);
```

```
build (2*v+1, m+1, tr);
          t[v]=t[2*v]+t[v*2+1];
          }
     }

void up(long long v, long long tl, long long tr, long long new_, long long pos)
     {
if (tl==tr) t[v]=new_;
else  {
int m=(tr+tl)/2;
if (pos<=m)
up(2*v, tl, m, new_, pos);
else up(2*v+1, m+1, tr, new_, pos);
            t[v]=t[2*v]+t[2*v+1];
          }
     }

long long sum (long long v, long long tl, long long tr, long long M)
     {

if(tl==tr)  return tl;
int m=(tl+tr)/2;
if (t[2*v]>M) return sum(v*2,tl, m, M);
else return sum(2*v+1, m+1, tr,M-t[2*v]);

     }

int main()

{  long long k, q, x, y, n, i;

cin>>n;
for (i=1; i<=n; ++i)
cin>>a[i];
        n++;
        a[n]=2000000000;
build(1, 1, n);
cin>>q;
for (i=0; i<q; ++i)
    {
cin>>k;
if (k==1) { cin>>x; cout<<sum(1, 1, n, x)-1<<»\n»;}
if (k==2) {cin>>x>>y; up(1, 1, n, y, x);}
    }
return 0;

}
```

Максимальна сума

Задано послідовність цілих чисел a_1, a_2, ..., a_n ($\mathbf{0 \le a_i \le 10^8}$, $\mathbf{2 \le n \le 10^5}$). До неї застосовуються операції двох типів:

Update:

Операція позначається символом «**U**», за яким слідують два цілі числа i та x.

***U i x*, $\mathbf{1 \le i \le n}$** та **$\mathbf{0 \le x \le 10^8}$**

Ця операція встановлює значення a_i рівним x.

Query:

Операція позначається символом «**Q**», за яким слідують два цілі числа ***i*** та ***j***.

***Q x y*, $\mathbf{1 \le x < y \le n}$**

Необхідно знайти такі ***i*** та ***j***, що $\boldsymbol{x \le i}$, $\boldsymbol{j \le y}$ та $\boldsymbol{i \ne j}$, для яких сума $a_i + a_j$ максимальна. Вивести значення суми $a_i + a_j$.

Вхідні дані

Перший рядок містить довжину послідовності ***n***. Наступний рядок містить n цілих чисел a_i. Наступний рядок містить кількість запитів q ($\boldsymbol{q \le 10^5}$). Далі q рядків описують операції, що виконуються на послідовності.

Вихідні дані

Для кожної *Query* операції вивести значення максимальної суми.

Вхідні дані	Вихідні дані
5 1 2 3 4 5 6 Q 2 4 Q 2 5 U 1 6 Q 1 5 U 1 7 Q 1 5	7 9 11 12

Розв'язання

Для побудови дерева в кожній вершині зберігатимемо два максимуми. У листку – це буде елемент масиву та $-\infty$. Для кожного наступного елемента шукатимемо два максимуми серед чотирьох значень.

```
#include<bits/stdc++.h>
#define MAXN 100005

using namespace std;
struct MM
{
     int m1,m2;
};
MM t[4*MAXN];

void build(int a[], int v, int tl, int tr)
{
     if(tl==tr) {t[v].m1=a[tl]; t[v].m2 = -1;}
     else{
              int tm=(tl+tr)/2;
              build(a,2*v,tl,tm);
              build(a,2*v+1,tm+1,tr);
              int x[4]={ t[2*v].m1, t[2*v].m2, t[2*v+1].m1, t[2*v+1].m2};
              sort(x,x+4);
              t[v].m1=x[3];
              t[v].m2=x[2];
     }
}

MM sum_build(int v,int tl,int tr,int l,int r)
{
     if(l>r) {
       MM temp; temp.m1=temp.m2=-1;
     return temp;
     }
     if(l==tl&&r==tr)
     {
     return t[v];
     }
     int tm=(tl+tr)/2;
     MM L = sum_build(2*v ,tl, tm,l, min(r,tm));
     MM R = sum_build(2*v+1,tm+1,tr , max (l,tm+1), r );
     int temp[]={L.m1,L.m2,R.m1,R.m2};
     sort(temp,temp+4);
     MM res;
     res.m1=temp[3];res.m2=temp[2];
     return res;
}

void update(int v,int tl,int tr,int pos,int new_val)
{
     if(tl==tr)
```

```
    {
    t[v].m1=new_val;
    t[v].m2=-1;
    }
    else
    {
    int tm = (tl+tr)/2;
    if(pos<=tm)
    update(2*v,tl,tm,pos,new_val);
    else
    update(2*v+1,tm+1,tr,pos,new_val);
    int temp[]={t[2*v].m1,t[2*v].m2,t[2*v+1].m1,t[2*v+1].m2};
    sort(temp,temp+4);
       t[v].m1=temp[3];
       t[v].m2 =temp[2];
    }
}

int main()
{

    int n=10,l,r,q,i;
    char c;
int a[MAXN]={1,2,4,3,5,4,3,2,3,4};
scanf(«%d»,&n);
for(i=0;i<n;i++)
scanf(«%d»,&a[i]);
build(a,1,0,n-1);
scanf(«%d»,&q);
for(i=0;i<q;i++)
 {
    scanf(« %c %d %d»,&c,&l,&r);
    if(c==’Q’)
    {
            MM res=sum_build(1,0,n-1,l-1,r-1);
            cout<<res.m1+res.m2<<endl;
    }
    else
    {
    update(1,0,n-1,l-1,r);
    }
 }
return 0;
}
```

Обчислювальна геометрія

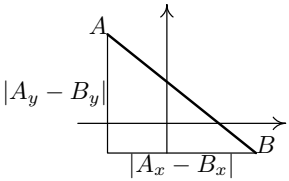
1. Теоретичний матеріал

1.1. Точки та вектори

Точки площини зазвичай задають у декартовій прямокутній системі у вигляді пар координат $(x; y)$. У програмах точки зазвичай подають таким типом записів:

```
type TPoint = record
  x, y: extended
end;
```

Наприклад, може бути змінна `A` типу `TPoint`; тоді до окремих її координат можна звернутися через `A.x` та `A.y`. Відповідно, з метою полегшення переходу між математичним і програмним записами, будемо і в суто математичних формулах використовувати позначення, наприклад, $(A_x; A_y)$ (замість більш уживаного на уроках математики $(x_A; y_A)$).



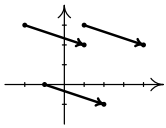
Відстань між точками A і B з координатами $(A_x; A_y)$ та $(B_x; B_y)$ (позначається $d(A, B)$, або, що те саме, $|AB|$) можна знайти за формулою:

$$d(A,B) = \sqrt{(A_x - B_x)^2 + (A_y - B_y)^2}. \quad (1)$$

Справді, за теоремою Піфагора: $(d(A,B))^2 = \underbrace{|A_x - B_x|^2}_{=(A_x-B_x)^2} + \underbrace{|A_y - B_y|^2}_{=(A_y-B_y)^2}$.

Вектор $\vec{a}$ можна означити як величину, що характеризується числовим значенням і напрямком.

У деяких задачах, особливо фізичних, буває важливо, де починається (до якої точки прикладений) вектор. Ми ж розглядатимемо переважно *вільні* вектори, які дозволено переносити в інше місце, важливо лише не змінювати довжину та напрям.



Наприклад, на рис. тричі показано один і той самий (вільний) вектор.

Є два стандартні, альтернативні один одному, способи задати вектор:

- Через довжину (позначатимемо або $|a|$, або l_a, це одне й те ж) і кут нахилу α

(Кут нахилу вектора рахують згідно означення тригонометричного кута, тобто у радіанах, від осі Ox, проти годинникової стрілки. Кути, що відрізняються один від одного на $\pm 2\pi$, $\pm 4\pi$ і т. д., однакові.)

- Через координати, вони ж проекції $(a_x; a_y)$.

Наприклад, вектор, тричі зображений на рис. вище, можна задати або як «координати (проекції) $a_x = 3$, $a_y = -1$», або «довжина $\sqrt{10}$, напрям $\operatorname{arctg} \frac{-1}{3} \approx -0{,}32$ (радіан)».

Подання вектора парою координат $(a_x; a_y)$ ***фактично однакове з поданням точки; отже, для подання (вільного) вектора можна використати той самий тип*** `TPoint`***, що для точок.***

За потреби, можна переходити від подання довжиною $|a|$ і напрямом α до подання координатами (a_x, a_y), за формулами

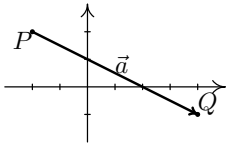
$$a_x = |a| \cdot \cos\alpha; \quad a_y = |a| \cdot \sin\alpha. \tag{2}$$

У зворотному напрямі —

$$|a| = \sqrt{a_x^2 + a_y^2}; \quad \alpha = \texttt{arctan2}(a_y, a_x). \tag{3}$$

(`Arctan2` — не математичне позначення, а функція мови програмування. Параметри цієї функції — координати вектора (спочатку y, потім x), результат — кут його нахилу ($-\pi < \alpha \leqslant \pi$). У FreePascal/Delphi ця функція називається `arctan2` і потребує підключення `uses Math`, у C/C++ називається `atan2` і потребує підключення `#include <math.h>`. Величезними перевагами цієї функції над очевидним виразом `arctan(y/x)` є: 1) вміє розрізнити протилежні кути, наприклад, $\texttt{arctan2}(3, 3) = \frac{\pi}{4}$, $\texttt{arctan2}(-5, -5) = -\frac{3\pi}{4}$, хоча $\frac{3}{3} = \frac{-5}{-5} = 1$; 2) працює при $x = 0$.)

Для будь-яких точок P та Q з координатами $(P_x; P_y)$ і $(Q_x; Q_y)$ можна побудувати вектор $\overrightarrow{PQ}$ з початком P і кінцем Q. Координати вектора $\overrightarrow{PQ}$ дорівнюють $(Q_x - P_x; Q_y - P_y)$.



$P_x = -2$, $P_y = 2$;
$Q_x = 4$, $Q_y = -1$;
$a_x = 6$, $a_y = -3$.

Аналогічно, якщо розмістити початок вектора $\vec{a}$ у точку P, то кінець вектора потрапить у точку з координатами $(P_x + a_x; P_y + a_y)$, де $(a_x; a_y)$ — координати вектора $\vec{a}$, $(P_x; P_y)$ — координати точки P.

Якщо сумістити (перенести в одну й ту ж точку площини) кінець вектора $\vec{a}$ і початок вектора $\vec{b}$, то *сумою векторів* $\vec{a}$ та $\vec{b}$ (позначається $\vec{a}+\vec{b}$) буде вектор, який починається у початку $\vec{a}$ і закінчується у кінці $\vec{b}$. Координати вектора-суми рівні $(a_x + b_x; a_y + b_y)$.

Добуток числа k на вектор $\vec{a}$ (записується $k \cdot \vec{a}$) — це вектор із координатами $(k \cdot a_x;\ k \cdot a_y)$. Якщо говорити про вектор як про напрямлений відрізок, то цей добуток має довжину $|k| \cdot l_a$, і при $k > 0$ вектор $k \cdot \vec{a}$ співнапрямлений з $\vec{a}$, а при $k < 0$ — протинапрямлений.

1.1.1. Програмна реалізація матеріалів цього підрозділу

Вищенаведені математичні поняття можна реалізувати підпрограмами:

```
(*Для утворення вектора по кінцю A й початку B.
Крім того, являє собою різницю векторів, де A - B = A + (-1)*B  *)
function minus (const A, B : TPoint) : TPoint;
Begin
  Result.x := A.x - B.x;
  Result.y := A.y - B.y;
End;

(*Для утворення точки кінця вектора
по точці початку A й вектору B.
Крім того, являє собою суму векторів A + B*)
function plus (const A, B : TPoint) : TPoint;
Begin
  Result.x := A.x + B.x;
  Result.y := A.y + B.y;
End;

(*Добуток числа на вектор*)
function multiply (k : extended ; const A : TPoint) : TPoint;
Begin
  Result.x := k*A.x;
  Result.y := k*A.y;
End;
```

Що всі ці підпрограми дають? Головним чином — можливість оперувати у програмі з точками і векторами як єдиними сутностями. Як наслідок — зручно знаходити потрібні точки/вектори.

Чи можна обходитися без таких підпрограм? У принципі, можна. Співвідношення аналітичної геометрії можна не кодувати мовою програмування, а ставитися до них як до рівнянь, розв'язувати їх олівцем на папері, отримувати прямі аналітичні формули вираження шуканих координат через відомі й писати у програмі вже їх. Часом це навіть призводить до значно коротших програм. Що ж, для деяких задач такий підхід має повне право на існування; особливо, якщо розв'язати треба лише одну геометричну задачу, і для неї виведення прямої аналітичної формули на папері виявляється досить простим.

Але коли є також інші геометричні задачі, ситуація змінюється: вже написані підпрограми принесуть користь також і в інших задачах/підзадачах, а от затрати часу на розв'язування рівнянь, аналітичні перетворення й виведення нестандартних прямих формул рідко коли приносять користь зразу для багатьох різних задач.

Інший вагомий аргумент на користь написання підпрограм, а не аналітичних виведень формул на папері полягає у тому, що навіть довга програмна реалізація може бути зручною для розуміння, якщо вона розщеплюється на *дуже прості* кроки, кожен із яких легко уявити і твердо переконатися у його правильності. Крім того, технічні помилки програми, яка крок за кроком виконує проміжні побудови (знаходить допоміжні вектори, точки й інші геометричні сутності), можна шукати засобами налагодження програм (debug-a). А якщо якась помилка трапилася десь невідомо де в аналітичному розв'язуванні системи рівнянь, маємо і складніші міркування/перетворення, і, як правило, повну відсутність технічних засобів, що могли б допомогти розібратися.

1.2. Скалярний добуток

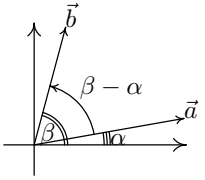
Скалярний добуток векторів $\vec{a}$ та $\vec{b}$ (позначається $\vec{a} \cdot \vec{b}$) можна визначити одним із двох способів:

- $|a| \cdot |b| \cdot \cos(\beta - \alpha)$, де $|a|$ та $|b|$ — довжини, α та β — кути нахилу;
- $a_x b_x + a_y b_y$.

Якщо вважати відомими формули (2) вираження координат вектора через кут і нахил, а також формулу вираження $\cos(\beta - \alpha)$, то доведення рівності двох наведених виразів зводиться до

$$\begin{gathered} a_x b_x + a_y b_y = \\ = (|a|\cos\alpha)(|b|\cos\beta) + (|a|\sin\alpha)(|b|\sin\beta) = \\ = |a|\cdot|b|\cdot\big(\cos\alpha\cos\beta + \sin\alpha\sin\beta\big) = \\ = |a|\cdot|b|\cdot\big(\cos\beta\cos\alpha + \sin\beta\sin\alpha\big) = \\ = |a|\cdot|b|\cdot\cos(\beta-\alpha). \end{gathered}$$

Утім, розуміння наведених перетворень корисне, але не є абсолютно обов'язковим.



Обов'язкові: 1) розуміння самого факту рівності $a_x b_x + a_y b_y = |a|\cdot|b|\cdot\cos(\beta-\alpha)$; 2) розуміння, що $\beta-\alpha$ являє собою *кут між векторами* $\vec{a}$ та $\vec{b}$.

Із цього (і факту, що косинус додатний для гострих кутів і від'ємний для тупих) слідує, що скалярний добуток:

- додатний тоді й тільки тоді, коли кут між векторами гострий;
- дорівнює нулю тоді й тільки тоді, коли вектори перпендикулярні;
- від'ємний тоді й тільки тоді, коли кут між векторами тупий.

Надалі, вважатимемо готовою функцію

```
(*Скалярний добуток*)
function scal_prod (const A, B : TPoint) : extended;
Begin
  Result := A.x*B.x + A.y*B.y;
End;
```

1.3. Орієнтована площа (косий добуток, псевдоскалярний добуток, векторний добуток для площини)

Важливу роль в обчислювальній геометрії на площині відіграє вираз

$$a_x b_y - a_y b_x \tag{4}$$

(який, зазвичай, не згадують у шкільному курсі алгебри/геометрії). На жаль, для нього нема єдиної загальноприйнятої назви; *усі* написані у підзаголовку варіанти «орієнтована площа», «косий добуток», «псевдоскалярний добуток» і «векторний добуток для площини» відповідають саме йому. Позначатимемо цю величину як $\vec{a}\times\vec{b}$ (зверніть увагу: для чисел «$6\cdot 7$» та «6×7» позначають абсолютно один і той

самий добуток, а для векторів «$\vec{a} \cdot \vec{b}$» та «$\vec{a} \times \vec{b}$» як правило, дають *різні* результати). Вважатимемо надалі, що у програмі є функція

```
(*Косий добуток, він же псевдоскалярний, він же
векторний добуток для площини, він же орієнтована площа*)
function cross_prod (const A, B : TPoint) : extended;
Begin
  Result := A.x*B.y - A.y*B.x;
End;
```

Провівши перетворення

$$\begin{gathered} a_x b_y - a_y b_x = \\ = (|a|\cos\alpha)(|b|\sin\beta) - (|a|\sin\alpha)(|b|\cos\beta) = \\ = |a| \cdot |b| \cdot \big(\cos\alpha\sin\beta - \sin\alpha\cos\beta\big) = \\ = |a| \cdot |b| \cdot \big(\sin\beta\cos\alpha - \sin\alpha\cos\beta\big) = \\ = |a| \cdot |b| \cdot \sin(\beta - \alpha) \end{gathered}$$

(аналогічні перетворенням для скалярного добутку), отримаємо добуток довжин векторів на *синус* кута між ними.

Із цього та факту, що синус додатний для кутів $0 < \varphi < \pi$ і від'ємний для кутів $(-\pi) < \varphi < 0$, слідує, що значення виразу $a_x b_y - a_y b_x$:

- додатне тоді й тільки тоді, коли напрям найкоротшого повороту від $\vec{a}$ до $\vec{b}$ додатний (проти годинникової стрілки);
- дорівнює нулю тоді й тільки тоді, коли вектори колінеарні (співнапрямлені або протинапрямлені);
- від'ємне тоді й тільки тоді, коли напрям найкоротшого повороту від $\vec{a}$ до $\vec{b}$ від'ємний (за годинниковою стрілкою).

Крім можливості класифікувати повороти за знаком, ця величина має зв'язок ще й з площею: добуток довжин сторін на синус кута між ними — це площа паралелограма, або ж подвоєна площа трикутника. Щоправда, площа не буває від'ємною, а досліджувана величина $a_x b_y - a_y b_x$ буває. Тож, щоб отримати площу $\triangle ABC$, у якому вершини A, B, C задані координатами, достатньо виконати такі кроки:

1) знайти вектор $\overrightarrow{AB}$ як $B - A$;
2) знайти вектор $\overrightarrow{AC}$ як $C - A$;

3) обчислити власне площу як

$$S_{\triangle ABC} = \left| \frac{\overrightarrow{AB} \times \overrightarrow{AC}}{2} \right|. \tag{5}$$

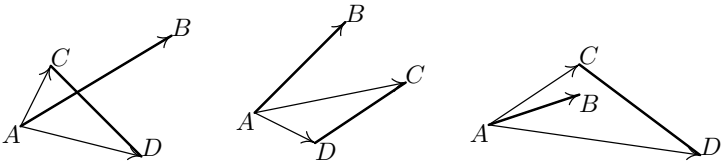
Цей спосіб знаходження площі дійсно працює, і (якщо відомі координати вершин трикутника, а не самі лише довжини сторін) має значні переваги над більш відомою формулою Герона $\sqrt{p(p-a)(p-b)(p-c)}$ — формула (5) і швидше обчислюється, і з нею значно рідше виникає проблема накопичення похибки (неточності обчислень).

Ще один цікавий факт, пов'язаний із (5), розглянемо у розд. 1.3.3.

1.3.1. Чи перетинаються відрізки — 1

Задано (координатами) чотири точки A, B, C, D, *щодо яких гарантовано, що ніякі три не лежать на одній прямій*. Чи перетинаються відрізки AB і CD?

Розглянемо вектори $\overrightarrow{AB}$, $\overrightarrow{AC}$ та $\overrightarrow{AD}$. Оскільки ніякі три точки не лежать на одній прямій, то $\overrightarrow{AB}$ не колінеарний ні $\overrightarrow{AC}$, ні $\overrightarrow{AD}$, і тому $\overrightarrow{AB} \times \overrightarrow{AC} \neq 0$, $\overrightarrow{AB} \times \overrightarrow{AD} \neq 0$. З рис. (a) та (b) видно, що коли точки C та D у різних півплощинах від прямої AB, то $\overrightarrow{AB} \times \overrightarrow{AC}$ та $\overrightarrow{AB} \times \overrightarrow{AD}$ різних знаків, а коли в одній — однакових.



При цьому не можна стверджувати, наприклад, $\overrightarrow{AB} \times \overrightarrow{AC} > 0$ `and` $\overrightarrow{AB} \times \overrightarrow{AD} < 0$, бо це так лише для конкретного випадку з рисунка; а от стверджувати «$\overrightarrow{AB} \times \overrightarrow{AC}$ та $\overrightarrow{AB} \times \overrightarrow{AD}$ різних знаків» — можна. Завдяки тому, що ми оголосили координати (поля `TPoint`) та результат функції `cross_prod` в типі `extended`, тут зручно застосувати математичну умову «два ненульові числа a, b різних знаків тоді й тільки тоді, коли $a \cdot b < 0$» (якби намагалися зробити те саме, маючи цілочисельні типи,

це було б неправильно, бо дуже часто траплялися б переповнення, внаслідок яких додатність чи від'ємність визначалася б неправильно.

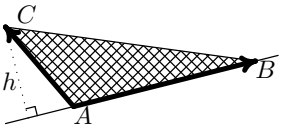
Рис. (c) показує, що можлива ситуація, коли C та D по різні боки від *прямої* AB, але *відрізки* AB і CD все ж не перетинаються. Тому умову перетину відрізків слід остаточно сформулювати, наприклад, так:

$$\left(\left(\overrightarrow{AB}\times\overrightarrow{AC}\right)\cdot\left(\overrightarrow{AB}\times\overrightarrow{AD}\right)<0\right)\texttt{and}\left(\left(\overrightarrow{CD}\times\overrightarrow{CA}\right)\cdot\left(\overrightarrow{CD}\times\overrightarrow{CB}\right)<0\right) \tag{6}$$

(одночасно і C та D по різні боки від AB, і A та B по різні боки від CD).

Насамкінець, така перевірка існування перетину відрізків працює *лише* завдяки тому, що в умові було гарантовано, що ніякі три з точок A, B, C, D не лежать на одній прямій. Коли гарантії нема, перевірка перетину відрізків стає набагато складнішою задачею, що потребує розгляду значно більшої кількості випадків.

1.3.2. Відстань від точки до прямої



Щоб знайти відстань між точкою C і прямою AB, розглянемо $\triangle ABC$ і виразимо (аналітичними формулами на папері, а не у програмі) його площу двома різними способами. З одного боку, вона згідно (5) дорівнює $\left|\frac{\overrightarrow{AB}\times\overrightarrow{AC}}{2}\right|$. З іншого, вона дорівнює $\frac{1}{2}\cdot d(A,B)\cdot h$ (де $d(A,B)=|AB|$ — довжина відрізка AB), тому що можна вважати AB основою, h висотою. Маємо

$$\left|\frac{\overrightarrow{AB}\times\overrightarrow{AC}}{2}\right|=\frac{1}{2}\cdot d(A,B)\cdot h;$$

домноживши на 2 (це додатна константа, тому її можна проносити крізь модуль) і поділивши на $d(A,B)$, остаточно отримуємо

$$h=\frac{\left|\overrightarrow{AB}\times\overrightarrow{AC}\right|}{d(A,B)}. \tag{7}$$

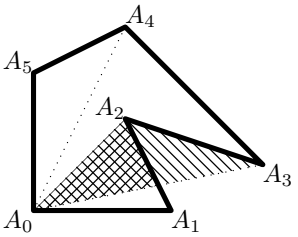
1.3.3. Площа простого многокутника

Многокутник на площині задано цілочисельними координатами N вершин. Потрібно знайти його площу. Многокутник простий, тобто його сторони не перетинаються і не дотикаються (за винятком сусідніх, у вершинах), але він *не обов'язково опуклий*.

(Приклад *не*опуклого простого многокутника див. на рис.)

Спочатку поміркуємо, що можна було б зробити, якби многокутник був опуклий. Наприклад, його можна було б розбити на трикутники, провівши всі можливі діагоналі з однієї й тієї ж вершини. Рахувати площу окремо взятого трикутника ми вміємо, і залишається тільки пододавати площі цих трикутників.

Спробуємо зробити так само (додати площі $A_0A_1A_2$, $A_0A_2A_3$, …, $A_0A_{n-2}A_{n-1}$) в неопуклому многокутнику. На перший погляд, усе сумно: заштриховані області включаються по кілька разів, хоча заштрихована у клітинку повинна потрапляти лише один раз, а заштрихована у смужку повинна взагалі не потрапляти.



Але виявляється, що при використанні *орієнтованих* площ трикутників усе гаразд! Наприклад, на рис. область, заштрихована у смужку, враховується спочатку як частина орієнтованої площі $\triangle A_0A_2A_3$ зі знаком «$-$», а потім як частина орієнтованої площі $\triangle A_0A_3A_4$ зі знаком «$+$», і в результаті не враховується (як і треба). Аналогічно, область, заштрихована у клітинку, додається двічі (як частина $\triangle A_0A_1A_2$ і $\triangle A_0A_3A_4$) й віднімається один раз (як частина $\triangle A_0A_2A_3$), і в результаті враховується один раз (знов-таки, як і треба).

Тобто, треба просто додати $N-2$ штук *орієнтованих* площ трикутників $A_0A_1A_2$, $A_0A_2A_3$, …, $A_0A_{n-2}A_{n-1}$, а *наприкінці* все-таки взяти модуль, бо сума орієнтованих площ — орієнтована площа; якщо обхід многокутника відбувається у від'ємному напрямку (за годинниковою стрілкою), то вона виявляється від'ємною.

(Зрозуміло, що наведений конкретний приклад не може доводити правильність для всіх можливих випадків; але тепер твердження стало більш-менш зрозумілим, а формальне доведення спирається на акуратне застосування аналогічних міркувань.)

Насамкінець, розглянутий спосіб дозволяє знайти площу, обмежену будь-яким простим многокутником, але *не* будь-якою замкненою ламаною: якби не було навіть гарантій про відсутність самоперетинів, цей метод не був би гарантовано правильним.

2. Задачі основного дня (14.10.2017)

Цей комплект задач доступний для on-line перевірки на сайті `ejudge.ckipo.edu.ua`, змагання № 60.

Задача A. «Четверта вершина прямокутника»

Знаючи координати трьох вершин прямокутника на координатній площині, визначте координати четвертої вершини. Сторони прямокутника *не* зобов'язані бути паралельні осям координат. Три вершини *не* обов'язково задані поспіль (пропущена вершина може бути як після них, так і де завгодно всере́дині).

Вхідні дані

В одному рядку записані шість чисел — координати трьох вершин прямокутника (спочатку x та y однієї вершини, потім x та y іншої, потім x та y ще однієї). Числові значення координат цілі, абсолютна величина (модуль) кожного не перевищує 100.

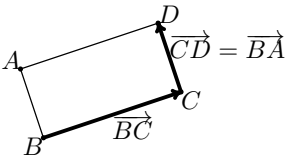
Результати

Виведіть в одному рядку через пропуск (пробіл) два числа́ — координати четвертої вершини прямокутника.

Приклади

input.txt або клавіатура (ст. вхід)	output.txt або екран (ст. вихід)
-2 3 4 3 4 -1	-2 -1
-1 2 0 0 6 3	5 5

Розбір



Для початку припустимо, ніби відомі верши́ни A, B, C, са́ме в порядку обходу прямокутника. Тоді, враховуючи $\overrightarrow{BA} = \overrightarrow{CD}$ (бо $ABCD$ прямокутник) та формули переходу між координатами точок і вільних векторів, можна знайти координати D як $B + \overrightarrow{BC} + \overrightarrow{CD} = B + \overrightarrow{BC} + \overrightarrow{BA} = B + (C-B) + (A-B)$. Тобто,

$$D = B + (C - B) + (A - B). \tag{8}$$

Але суттєва складова задачі в тому й полягає, що пропущеною може бути якась вершина всере́дині: тоді вхідні дані виявляються не у порядку обходу, й формула з попереднього абзацу перестає бути правильною. Що ж, доведеться написати трохи складніший код, який поєднуватиме і ту формулу, і деякий перебір випадків.

Оскільки задані у вхідних даних вершини гарантовано є трьома з чотирьох вершин прямокутника, вони гарантовано утворюють прямокутний трикутник. Яка з його вершин має прямий кут — наперед не відомо, але ми можемо це дізнатися, перебравши можливі варіанти: якщо $\overrightarrow{AB} \cdot \overrightarrow{AC} = 0$, то прямим є кут при вершині A; інакше, якщо $\overrightarrow{CA} \cdot \overrightarrow{CB} = 0$, то прямим є кут при вершині C; якщо жоден із попередніх варіантів не був вибраний, то при вершині B. (Де «$\cdot$» — скалярний добуток (розд. 1.2); замість перевірки за скалярними добутками в принципі можна робити якусь іншу перевірку, чи прямий кут, як-то «чи виконується теорема Піфагора»; але формули скалярного добутку кращі тим, що простіші.)

Можна піти шляхом переробки формули (8) під кожен випадок, як-то «якщо $\overrightarrow{AB} \cdot \overrightarrow{AC} = 0$, то $D = A + (C-A) + (B-A)$» (і так усі випадки). Але це поганий підхід, бо відбуваються повтори (зі змінами!) коду, відповідного (8), і в цьому можна легко заплутатися; краще винести обчислення згідно (8) у підпрограму і викликати її з різних гілок розгалуження з різним порядком аргументів.

Задача B. «Трикутник і точка»

Задано (координатами вершин) трикутник ABC і точку D. Визначте розміщення точки відносно трикутника, а са́ме: виведіть «`In`», якщо точка лежить строго всере́дині трикутника; «`Edge`», якщо точка лежить на стороні; «`Vertex`», якщо точка лежить на вершині; «`Out`», якщо точка лежить поза трикутником.

Вхідні дані

Чотири рядки по 2 цілих числа, що не перевищують за модулем 10 000 000.

Перші три рядки — координати вершин трикутника A, B, C. Четвертий — координати точки O.

Результати

Відповідь до задачі («`In`», «`Edge`», «`Vertex`» або «`Out`»).

Приклад

input.txt або клавіатура (ст. вхід)	output.txt або екран (ст. вихід)
-2 -2 3 1 0 1 0 0	In

Розбір

Якщо сума площ $S_{\triangle OAB}$, $S_{\triangle OBC}$ і $S_{\triangle OCA}$ перевищує $S_{\triangle ABC}$, точка лежить поза трикутником. Якщо ж сума перших трьох площ дорівнює четвертій, то перевіримо, чи не дорівнюють нулю деякі з перших трьох площ. Якщо жодна не дорівнює — точка строго всередині трикутника, якщо нулю дорівнює одна площа — точка на стороні, якщо дві — точка O співпадає з однією з вершин A, B або C.

Примітка. Порівнювати треба суму «звичайних» площ, а не орієнтованих. Сума орплощ завжди дорівнює орплощі ABC, незалежно від розміщення точки O.

Задача C. «Спільні дотичні — 1»

Як відомо, дотичною до кола є пряма, що має рівно одну спільну точку з цим колом. Можлива ситуація, коли одна й та сама пряма є дотичною відразу до двох кіл. Тоді вона називається спільною дотичною. Напишіть програму, яка знаходитиме кількість різних спільних дотичних для заданих двох кіл. При виведенні врахуйте стародавню традицію приписувати числу 7 значення «багато». Тобто, коли кількість спільних дотичних строго більша 6, незалежно від справжньої кількості виводьте 7.

Вхідні дані

Шість цілих чисел, розділених пропусками (пробілами), $X1$, $Y1$, $R1$, $X2$, $Y2$, $R2$ — відповідно координати центра і радіуси 1-го і 2-го кола. Для всіх координат абсолютна величина (модуль) не перевищує мільйон. Для обох радіусів значення у межах від 1 до мільйона (обидві межі включно).

Результати

Програма виводить єдине число — шукану кількість, з урахуванням згаданої стародавньої традиції.

Приклад

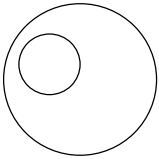
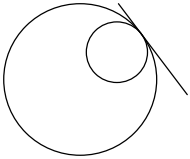
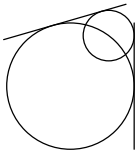
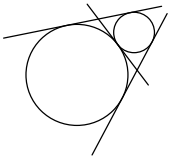
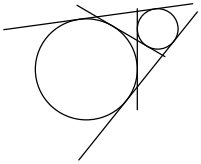
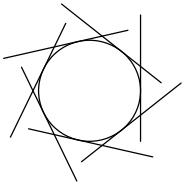
input.txt або клавіатура (ст. вхід)	output.txt або екран (ст. вихід)
20 0 4 50 0 10	4

Розбір

Легко придумати ситуації, коли два кóла мають 2 чи 4 спільні дотичні; ситуації, коли їх 0, 1 або 3, придумуються дещо важче; може здатися, ніби строго більше, ніж 4, взагалі не буває. Але насправді такий випадок (один) все-таки є: `(X1=X2) and (Y1=Y2) and (R1=R2)`, тобто кóла повністю однакові. Тоді абсолютно будь-яка дотична є спільною, і кількість виявляється нескінчéнною; це й треба позначати числом 7.

Тому для розв'язування конкретно цієї задачі абсолютно не потрібні ні тип `TPoint`, ні вектори взагалі. Потрібно *лише* вибрати, який із розглянутих випадків має місце, а для цього потрібні *лише* радіуси кіл $R1$,

$R2$ та відстань між центрами кіл $d = \sqrt{(X1 - X2)^2 + (Y1 - Y2)^2}$. Для зручності подальших викладок забезпечимо $R1 \geqslant R2$ (якщо це не так, досить обміняти їх місцями).

0 спільних дотичних:	1 спільна дотична:	2 спільні дотичні:
		
одне коло повністю всере́дині іншого	ко́ла дотикаються внутрішнім чином	ко́ла перетинаються
3 спільні дотичні:	**4 спільні дотичні:**	**∞ спільних дотичних:**
		
ко́ла дотикаються зовнішнім чином	ко́ла не перетинаються	ко́ла повністю однакові

- Якщо $(d = 0)$ and $(R1 = R2)$, відповідь 7 (насправді, ∞);
- інакше, якщо $d < R1 - R2$, відповідь 0;
- інакше, якщо $d = R1 - R2$, відповідь 1;
- інакше, якщо $R1 - R2 < d < R1 + R2$, відповідь 2;
- інакше, якщо $d = R1 + R2$, відповідь 3;
- інакше (при $d > R1 + R2$), відповідь 4.

Задача D. «Спільні дотичні — 2»

Як відомо, дотичною до кола є пряма, яка має рівно одну спільну точку з цим колом. Можлива ситуація, коли одна й та сама пряма є дотичною відразу до двох кіл. Тоді вона називається спільною дотичною. Напишіть програму, яка знаходитиме спільні дотичні для заданих

двох кіл. При виведенні врахуйте стародавню традицію приписувати числу 7 значення «багато». Тобто, коли кількість спільних дотичних строго більша 6, незалежно від справжньої кількості виводьте будь-які сім з усіх можливих спільних дотичних.

Вхідні дані

Шість цілих чисел, розділених пропусками (пробілами), $X1$, $Y1$, $R1$, $X2$, $Y2$, $R2$ — відповідно координати центра і радіуси 1-го і 2-го кола. Для всіх координат абсолютна величина (модуль) не перевищує мільйон. Для обох радіусів значення у межах від 1 до мільйона (обидві межі включно).

Результати

Програма виводить у першому рядку єдине число K — кількість шуканих спільних дотичних (з урахуванням згаданої стародавньої традиції); далі повинно йти рівно K рядків, кожен із яких повинен містити чотири дійсні числа́ — координати двох різних точок із відповідної дотичної (спочатку x- та y-координати однієї точки, потім x- та y-координати іншої).

Відповідь зараховуватиметься, коли виконуватимуться всі вимоги:

1. Кількість спільних дотичних знайдено правильно (з урахуванням вказаної стародавньої традиції).
2. Кожна дотична описується двома помітно різними точками (відстань між двома точками, що задають одну дотичну, не менша 1).
3. Пряма, що проходить через кожну з пари точок, або справді є дотичною (має рівно одну спільну точку з колом), або є добрим наближенням до дотичної (або проходить поза колом на відстані не більш як 10^{-6} від нього, або заходить усере́дину кола так, що довжина частини цієї прямої всере́дині цього кола не перевищує однієї мільйонної від радіуса цього кола).
4. Кожен із K рядків, що описують дотичні, описує свою власну дотичну, відмінну від інших; формально кажучи, якщо взяти j-й та k-й рядки ($j \neq k$), де дотичні задані точками A_j, B_j та A_k, B_k, то повинно виконуватися

$$((dist(A_j, A_kB_k) \geqslant 0{,}1) \texttt{ or } (dist(B_j, A_kB_k) \geqslant 0{,}1))$$
$$\texttt{and}$$
$$((dist(A_k, A_jB_j) \geqslant 0{,}1) \texttt{ or } (dist(B_k, A_jB_j) \geqslant 0{,}1)),$$

де $dist$ — відстань від точки до прямої, що рахується уздовж перпендикуляра; смисл усього виразу — хоча б одна з двох точок, які задають пряму, знаходиться на відстані хоча б 0,1 від іншої прямої.

Приклад

input.txt або клавіатура (ст. вхід)
`20 0 4 50 0 10`
output.txt або екран (ст. вихід)

`4`
`48 -9.79795897113 19.2 -3.91918358845`
`48 9.79795897113 19.2 3.91918358845`
`45.3333333333 -8.84433277428 21.8666666667 3.53773310971`
`45.3333333333 8.84433277428 21.8666666667 -3.53773310971`

Розбір

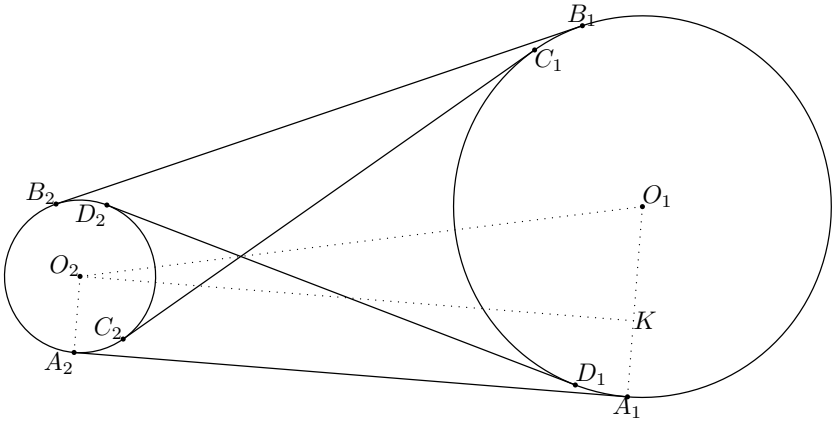
Очевидно, задача повністю включає в себе попередню. Щодо решти наведемо лише скорочені вказівки, залишивши значну частину цієї об'єктивно довгої й нудноватої задачі читачам.

Щоб знайти спільну дотичну, коли вона єдина:

1. Визначимо (через `arctan2` вектора `O2-O1`, де `O1` та `O2` — центри кіл) тригонометричний кут прямої, що з'єднує центри кіл.
2. Відступивши у цьому напрямку від точки $O1$ на відстань $R1$, знайдемо точку спільного дотикання — її ж можна узяти за одну з точок, через які проходить шукана дотична.
3. Додавши $\frac{\pi}{2}$ до кута з п. 1, отримаємо тригонометричний кут дотичної.
4. Відступимо від точки, знайденої у п. 2, у напрямку, знайденому у п. 3, на будь-яку розумну відстань, наприклад, $R1$.

Практично такі самі дії треба виконати також для знаходження спільної дотичної, що проходить через точку зовнішнього дотику (у випадку, коли всього дотичних 3). Якщо кóла цілком однакові, то дії теж можуть бути схожими, лише замість знаходження одного конкретного тригонометричного кута O_1O_2 слід узяти 7 довільних різних напрямків.

Залишаються тільки спільні дотичні, які не проходять через точку дотику кіл (усі 4, коли їх 4; обидві, коли їх 2; решта 2, коли їх 3).



Розглянемо пошук A_1 і A_2. Оскільки A_1A_2 — дотична, $\angle O_1A_1A_2$ і $\angle O_2A_2A_1$ прямі. Точку K побудуємо (на папері, а не в алгоритмі) як основу перпендикуляра, опущеного з O_2 на O_1A_1. Тоді $O_2KA_1A_2$ — прямокутник, а $\triangle O_2O_1K$ — прямокутний, і у нього відомі довжини гіпотенузи і катета O_1K — вони рівні $|O_1O_2|$ і $R_1 - R_2$ відповідно. (Нагадаємо, ми вважаємо забезпеченим $R_1 \geqslant R_2$; якщо це не так, то починаємо з того, що обмінюємо місцями повністю всі дані кіл.) Звідси, позначивши через α величину кута $\angle O_1O_2K$, отримуємо

$$\alpha = \arcsin \frac{R_1 - R_2}{|O_1O_2|}. \tag{9}$$

Кут нахилу вектора $\overrightarrow{O_2O_1}$ (позначимо його φ) теж, по суті, відомий (як `arctan2` вектора `O2-O1`). Тоді кути нахилу векторів O_2A_2 та O_1A_1 рівні $\varphi - \alpha - \frac{\pi}{2}$, векторів O_2B_2 та O_1B_1 рівні $\varphi + \alpha + \frac{\pi}{2}$. Тепер нескладно вирахувати декартові координати цих векторів, а по них — координати точок A_1, A_2, B_1 і B_2.

Неважко переконатися, що всі ці міркування застосовні як у випадку 4-х дотичних, так і у випадках 3-х чи 2-х.

У випадку 4-х дотичних для знаходження C_1, C_2, D_1 і D_2 повторюються майже ті ж міркування, тільки довжина катета дорівнює $R_1 + R_2$.

Задача E. «Годинник на сканері»

Секундна стрілка годинника переміщується стрибками, тобто упродовж секунди нерухома, а потім дуже швидко повертає на $\frac{1}{60}$ повного оберту. Стрілка являє собою тонкий відрізок довжини d мм, що виходить із центра годинника. Годинник поклали на сканер, орієнтувавши звичайним чином (позначка «12» згори) й підібрали параметри сканування так, що:

1. Сканування запускається відразу після того, як секундна стрілка виконала черговий стрибок і почала показувати s секунд.
2. Область сканування вибрана як квадрат розмірами $2d$ мм $\times$ $2d$ мм так, що вона в точності охоплює круг, який покриває секундна стрілка.
3. Сканер за (кожну) 1 с встигає отримати прямокутне зображення висотою рівно k мм та шириною $2d$ мм (тобто усієї області сканування).
4. Роздільна здатність сканера досить висока, щоб можна було знехтувати дискретністю зображень усере́дині кожної k-міліметрової смужки й рахувати відстані за звичайними геометричними формулами.

Знайдіть сумарну довжину зображень секундної стрілки в отриманій картинці (вважаючи, що зображення інших елементів годинника не створюють проблем).

Вхідні дані

Три цілі числа: k (висота області, яку сканують за 1 с), d (довжина стрілки) та s (скільки секунд почала показувати стрілка у момент початку сканування).

$1 \leqslant k \leqslant 50$; $0{,}75k \leqslant d \leqslant 100k$; відношення $\frac{d}{k}$ гарантовано не є цілим числом; $0 \leqslant s < 60$.

Результати

Вивести єдине дійсне число l — сумарну довжину зображень секундної стрілки.

Відповідь буде зарахована, якщо відносна або абсолютна похибка (хоча б одна з них) не перевищить 10^{-6}.

Приклад

input.txt або клавіатура (ст. вхід)	output.txt або екран (ст. вихід)
36 90 10	103.994544

Розбір

Для розв'язання задачі досить промоделювати кожну секунду окремо: знайти координати обох кінців стрілки; знайти прямокутник, що сканується упродовж цієї секунди; знайти перетин цієї стрілки й цього прямокутника (або з'ясувати, що він порожній). Оскільки гарантовано $0{,}75 \leqslant \frac{d}{k} \leqslant 100$, це займе не більше $2 \times 100 = 200$ ітерацій, що для комп'ютера зовсім мало. Умова, що $\frac{d}{k}$ не є цілим числом, робить неможливою ситуацію, коли горизонтальна стрілка (яка показує 15 чи 45 секунд) опиняється якраз на межі прямокутника, який сканують у відповідну секунду, а отже, не виникає питання, треба чи не треба враховувати те, що якраз на межі сканування.

Оскільки в задачі не задано систему координат, введемо її самі так, щоб початок координат був у центрі годинника; таким чином, один із кінців стрілки завжди матиме координати $(0; 0)$. Напрямки осей введемо звичайним чином (Ox праворуч, Oy нагору), бо це необхідно, щоб зберегти правильність формул (2) з розд. 1.1.

Коли секундна стрілка годинника показує t секунд, її тригонометричний кут становить $\frac{15-t}{30}\pi$. (Доведення: рух стрілки відбувається у від'ємному в тригонометричному смислі напрямку; повний оберт 2π рівномірно поділений між 60 секундами, звідси $\frac{-2\pi}{60} = -\frac{t}{30}\pi$; 0 секунд відповідають положенню вертикально вгору, тобто $+\frac{\pi}{2} = +\frac{15}{30}\pi$.)

Ураховуючи два попередні абзаци, кінці стрілки, що показує t секунд, матимуть координати $(0; 0)$ та $(d \cdot \cos\frac{15-t}{30}\pi;\ d \cdot \sin\frac{15-t}{30}\pi)$. Залишається знайти перетин цієї стрілки з поточним прямокутником сканування.

Оскільки поточний прямокутник сканування легше виражати не через той час, який показує годинник, а через той час, який минув від початку сканування, замінимо в усіх раніше введених формулах t на $s + \tau$, де s — це s зі вхідних даних, тобто скільки секунд показував годинник

на початку, а τ — скільки секунд минуло від початку сканування; ця величина починається з 0, далі 1, 2, …, $\lfloor \frac{2d}{k} \rfloor$.

Тоді можна виразити такі y-координати:

$$\text{кінець стрілки } y_h(\tau) = d \cdot \sin\Big(\frac{15 - (s + \tau)}{30}\pi\Big); \quad (10)$$

$$\text{низ стрілки } y_{h\,\min}(\tau) = \min(0, \underbrace{y_h(\tau)}_{(10)}); \quad (11)$$

$$\text{верх стрілки } y_{h\,\max}(\tau) = \max(0, \underbrace{y_h(\tau)}_{(10)}); \quad (12)$$

$$\text{низ прямокутника } y_{f\,\min}(\tau) = d - \tau \cdot k; \quad (13)$$

$$\text{верх прямокутника } y_{f\,\min}(\tau) = d - (\tau + 1) \cdot k. \quad (14)$$

y-проекція перетину (спільної частини) стрілки й прямокутника, що сканується, одночасно більша-рівна і $\underbrace{y_{h\,\min}(\tau)}_{(11)}$, і $\underbrace{y_{f\,\min}(\tau)}_{(13)}$; аналогічно, ця y-проекція одночасно менша-рівна і $\underbrace{y_{h\,\max}(\tau)}_{(12)}$, і $\underbrace{y_{f\,\max}(\tau)}_{(14)}$. Отже, варто знайти

$$y_{\max\,\min}(\tau) = \max\big(\underbrace{y_{h\,\min}(\tau)}_{(11)}, \underbrace{y_{f\,\min}(\tau)}_{(13)}\big); \quad (15)$$

$$y_{\min\,\max}(\tau) = \min\big(\underbrace{y_{h\,\max}(\tau)}_{(12)}, \underbrace{y_{f\,\max}(\tau)}_{(14)}\big). \quad (16)$$

У випадку $\underbrace{y_{\max\,\min}(\tau)}_{(15)} > \underbrace{y_{\min\,\max}(\tau)}_{(16)}$ конкретно у секунду τ прямокутник, що сканується, не має жодної спільної точки зі стрілкою, нічого додавати в результат не треба.

У випадку $\underbrace{y_{\max\,\min}(\tau)}_{(15)} < \underbrace{y_{\min\,\max}(\tau)}_{(16)}$ застосуємо пропорцію: уся стрілка довжиною d займала проміжок від $\underbrace{y_{h\,\min}(\tau)}_{(11)}$ до $\underbrace{y_{h\,\max}(\tau)}_{(12)}$, а фактично задіяним (видимим у прямокутнику, який сканують) є лише проміжок

від $\underbrace{y_{\max\min}(\tau)}_{(15)}$ до $\underbrace{y_{\min\max}(\tau)}_{(16)}$; отже, додати треба $\frac{y_{\min\max}(\tau)-y_{\max\min}(\tau)}{y_{h\,\max}(\tau)-y_{h\,\min}(\tau)}\cdot d$.

У випадку $\underbrace{y_{\max\min}(\tau)}_{(15)} = \underbrace{y_{\min\max}(\tau)}_{(16)}$ слід перевірити, чи стрілка похила або вертикальна (тоді перетин формально існує, але одноточковий і не впливає на сумарну довжину), чи горизонтальна (тоді треба додати повну довжину стрілки d).

Задача F. «Відстань від точки до відрізка»

Дано точку P з координатами Px, Py та відрізок AB, кінці якого мають координати Ax, Ay та Bx, By. Відрізок гарантовано не вироджений, тобто A та B — різні точки.

Напишіть програму, яка знаходитиме відстань між точкою P та відрізком AB.

Примітка. Відстань між точкою та відрізком слід трактувати згідно зі стандартним означенням відстані між точкою та складним геометричним об'єктом: якщо точка належить цьому об'єкту, відстань рівна нулю; якщо не належить, відстань рівна довжині найкоротшого з можливих відрізків, для яких одним із кінців є дана точка, а інший кінець належить цьому об'єкту.

Вхідні дані

Слід прочитати зі стандартного входу (клавіатури), у форматі Px Py Ax Ay Bx By (в одному рядку). Всі координати цілі й не перевищують за модулем 10000.

Результати

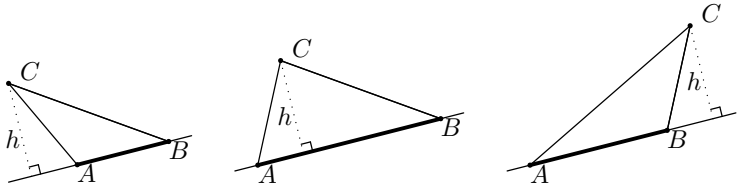
Вивести єдине число — знайдену відстань від точки до відрізка. Виводити можна хоч у експоненційній формі, хоч стандартним десятковим дробом. Результат зараховується, коли похибка (абсолютна або відносна, тобто хоча б одна з них) не перевищує 10^{-6}.

Приклад

input.txt або клавіатура (ст. вхід)	output.txt або екран (ст. вихід)
0 4 2 3 2 5	2.0

Розбір

Задача спирається на розд. 1.3.2: там описано, як шукати відстань між точкою C і *прямою* AB. Розглянемо (як і у розд. 1.3.2) $\triangle ABC$, виділяючи AB як основу і опускаючи на AB висоту з C.



Із цих рисунків і наведеного в умові задачі означення відстані між точкою і протяжним об'єктом ясно, що може бути три випадки:

- якщо $\angle CAB$ тупий, то найкоротшим із можливих відрізків від точки C до відрізка AB є відрізок CA;
- якщо $\angle CBA$ тупий, то найкоротшим із можливих відрізків від точки C до відрізка AB є відрізок CB;
- якщо жоден із кутів $\angle CAB$ чи $\angle CBA$ не є тупим, то найкоротшим із можливих відрізків від точки C до відрізка AB є висота, і її слід обчислити згідно формули (7) з розд. 1.3.2.

Тобто, залишилося тільки з'ясувати, який із цих випадків має місце. Наприклад, це можна зробити за допомогою скалярних (розд. 1.2) добутків: $\angle CAB$ тупий $\Leftrightarrow \overrightarrow{AB} \cdot \overrightarrow{AC} < 0$, $\angle CBA$ тупий $\Leftrightarrow \overrightarrow{BA} \cdot \overrightarrow{BC} < 0$.

(У принципі, можливі й альтернативні способи перевірки, наприклад: $\angle CAB$ тупий $\Leftrightarrow |BC|^2 > |AC|^2 + |AB|^2$. Але навряд чи вони об'єктивно кращі й простіші за перевірку скалярних добутків.)

Зауважимо, що тип $\angle ACB$ (гострий/прямий/тупий) ніяк не впливає на розв'язок. Ще зауважимо, що випадок прямого $\angle CAB$ немає потреби розглядати окремо: його можна віднести хоч до випадку тупого кута, хоч до випадку гострого, і так і так буде правильно, бо при прямому $\angle CAB$ висота дорівнює катету. З $\angle CBA$ аналогічно.

Задача G. «Площа простого многокутника»

Многокутник на площині задано цілочисельними координатами N вершин. Потрібно знайти його площу.

Многокутник простий, тобто його сторони не перетинаються і не дотикаються (за винятком сусідніх, у вершинах), але він не обов'язково опуклий.

Вхідні дані

Слід прочитати зі стандартного входу (клавіатури). У першому рядку задано кількість вершин N ($1 \leqslant N \leqslant 50000$). У наступних N рядках записані пари чисел — координати вершин. Сторони многокутника — відрізки між 1-ою і 2-ою, 2-ою і 3-ою, ..., $(N-1)$-ою і N-ою, N-ою і 1-ою вершинами. Значення координат — цілі числа, не перевищують за модулем мільйон.

Результати

Вивести єдине число — знайдену площу многокутника. Виводити можна хоч у експоненційній формі, хоч стандартним десятковим дробом. Результат зараховується, коли похибка (абсолютна або відносна, тобто хоча б одна з них) не перевищує 10^{-6}.

Приклад

input.txt або клавіатура (ст. вхід)	output.txt або екран (ст. вихід)
4 0 4 0 0 3 0 1 1	3.5

Задача повністю розібрана у розд. 1.3.3.

Задача H. «Чи є маршрут приємним?»

Туристу набридло подорожувати уздовж координатної осі, тому він вирішив помандрувати ще по координатній площині. Він розпочинає зі

своєї бази в точці A_1 з координатами (x_1, y_1), рухається найкоротшим маршрутом до визначної пам'ятки A_2 з координатами (x_2, y_2), далі, не зупиняючись, рухається найкоротшим маршрутом до визначної пам'ятки A_3 з координатами (x_3, y_3), і так далі. Дійшовши до останньої визначної пам'ятки A_n з координатами (x_n, y_n), він, не зупиняючись, рухається до своєї бази. Турист вважає свій маршрут *неприємним*, якщо існує така пряма, що він уздовж неї не рухався, і разом з тим перетинав її строго більше двох разів. Якщо маршрут не є неприємним, турист вважає його *приємним*.

Турист вважає, що перетинав пряму, якщо в деякий момент часу перебував у одній півплощині відносно неї, а через деякий проміжок часу — в іншій півплощині (сама пряма не належить жодній із півплощин).

Напишіть програму, яка, прочитавши описи кількох маршрутів, визначить, чи приємний кожен із них.

Вхідні дані

Програма повинна прочитати спочатку кількість маршрутів K ($2 \leqslant \leqslant K \leqslant 12$), потім K однотипних блоків, кожен із яких описує маршрут. Кожен блок опису маршруту починається числом n ($2 \leqslant n \leqslant 98765$), далі йдуть n пар цілих чисел, що не перевищують 10^8 за абсолютною величиною — координати $x_1\ y_1\ x_2\ y_2\ \ldots\ x_n\ y_n$. Усі числа всіх маршрутів записані в одному рядку й розділені одинарними пробілами. Сумарна кількість усіх вершин усіх маршрутів, які програма має обробити за один запуск, не перевищуватиме 123456.

Результати

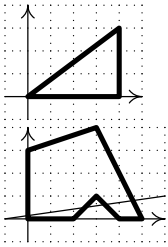
Програма повинна вивести в один рядок K розділених пробілами нулів та/або одиниць, які позначають, приємними (1) чи неприємними (0) були відповідні маршрути.

Приклад

input.txt або клавіатура (ст. вхід)
2 3 0 0 4 0 4 3 7 0 3 0 0 2 0 3 1 4 0 5 0 3 4
output.txt або екран (ст. вихід)
1 0

Примітка

Многокутники з прикладу вхідних даних зображені на рисунку. У першому випадку неможливо провести таку пряму, щоб турист уздовж неї не рухався, але перетинав її строго більше двох разів. У другому випадку наведено один із багатьох можливих прикладів са́ме такої прямої.



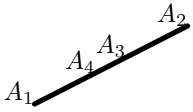
Розбір

Поняття «приємного маршруту», очевидно, слід перетворити в деяке стандартніше. Але важливо при цьому нічого не змінити в постановці задачі. Наприклад, *майже* правильно сказати, що «приємним маршрутом» є «обхід опуклого многокутника» — але не зовсім! Зокрема, будь-який маршрут при $n = 2$ (який виходить із «бази» в єдину «пам'ятку» й тут же уздовж того са́мого відрізка повертається на «базу») «приємний», хоча й не є опуклим многокутником. Так само важливо не допустити помилкового уявлення, ніби заданий у вхідних даних маршрут обов'язково задовольняє якісь додаткові вимоги, про які в умові нічого не сказано. Насправді маршрут може взагалі не бути многокутником, маючи повтори вершин (деякі послідовні чи не послідовні $(x_j; y_j)$ та $(x_k; y_k)$, при $k \neq j$, розміщені в одній і тій самій точці площини). Крім того, нема ніяких причин істотно розрізняти «пам'ятки» від «бази»; все називатимемо однаково: вершини.

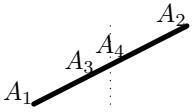
Почнемо розв'язок з того, що при читанні пропускатимемо вершини, розміщені там само (в тій самій точці площини), що й попередня; сюди слід включити також випадок, коли остання вершина розміщена там само, де перша (тоді слід видалити будь-яку з них, наприклад, останню); але якщо якісь інші пари вершин (розміщені у маршруті не підряд) теж розміщені в одній точці площини, то з цим нічого робити не будемо (взагалі це не перевірятимемо). Надалі вважатимемо за кількість вершин n не те n, котре було задане у вхідних даних, а те (чи то таке саме, чи то зменшене), яке утворилося після відкидань повторів однакових координат. Така обробка не змінює остато́чну відповідь (приємний маршрут чи ні), зате після неї жоден із векторів $\overrightarrow{A_1A_2}$, $\overrightarrow{A_2A_3}$, …, $\overrightarrow{A_{n-1}A_n}$, $\overrightarrow{A_nA_1}$ не є нульовим. (Нагадаємо, що нульовий вектор, або нуль-вектор, має нульову довжину (кінець дорівнює початку).)

Нуль-вектор — єдиний (вільний) вектор, для якого не визначений тригонометричний кут нахилу; позбувшися нуль-векторів у послідовності $\overrightarrow{A_1A_2}$, $\overrightarrow{A_2A_3}$, …, $\overrightarrow{A_{n-1}A_n}$, $\overrightarrow{A_nA_1}$, можна працювати з кутами нахилу та ввести поняття «кут повороту у вершині». Неформально його можна описати як *«якщо рухалися в напрямку $\overrightarrow{A_{i-1}A_i}$, то на скільки треба повернути, щоб після повороту рухатися в напрямку $\overrightarrow{A_iA_{i+1}}$?»*.

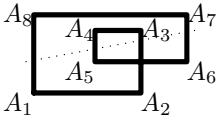
Перевірка № 1. Розглянемо кути повороту другого многокутника з прикладів з умови. У ньому 6 із 7 кутів повороту додатні (проти годинникової стрілки, ліворуч), один від'ємний (за годинниковою стрілкою, праворуч). І маршрут «неприємний» са́ме через той «зубець» $(3;1)$, який і має протилежний знак повороту. Все це схиляє до того, ніби маршрут слід перевіряти за правилом: *«Якщо серед косих добутків $\overrightarrow{A_nA_1} \times \overrightarrow{A_1A_2}$, $\overrightarrow{A_1A_2} \times \overrightarrow{A_2A_3}$, …, $\overrightarrow{A_{i-1}A_i} \times \overrightarrow{A_iA_{i+1}}$, …, $\overrightarrow{A_{n-2}A_{n-1}} \times \overrightarrow{A_{n-1}A_n}$, $\overrightarrow{A_{n-1}A_n} \times \overrightarrow{A_nA_1}$ є як додатні, так і від'ємні, то маршрут неприємний»*. Але таке правило *не* є *критерієм*. Наприклад:



тут лише нулі, маршрут приємний



тут теж лише нулі, але маршрут неприємний (див. A_2A_3, A_3A_4, A_4A_1)



тут усі повороти в одну сторону, але маршрут неприємний

Тобто, аналізувати *лише напрями* поворотів за косими добутками — замало; вся ця «перевірка № 1» може бути складовою алгоритму, але потребу́є також інших складових. У принципі, відомі різні варіанти, які саме додаткові перевірки можна додати.

Перевірка № 2. Розглянемо проекції вершин на вісі, тобто окремо послідовність $A_{1\cdot x}$, $A_{2\cdot x}$, …, $A_{n\cdot x}$, окремо $A_{1\cdot y}$, $A_{2\cdot y}$, …, $A_{n\cdot y}$. Для всіх приємних маршрутів, для кожної з цих послідовностей справедливо: всю послідовність можна розділити на $\leqslant 3$ проміжки монотонності (в межах яких послідовність або лише не зростає, або лише не спадає). Іншими словами: у приємному маршруті може лише зростати, може лише спадати, може спочатку зростати, потім спадати, може навіть

спочатку спадати, потім зростати, потім спадати; але якщо мінімальна кількість проміжків строго більша, ніж три, то маршрут неприємний.

Ця перевірка № 2 сама по собі теж не гарантує правильності визначення, чи приємний маршрут; але взяті разом перевірки № 1 та № 2 гарантують. (Автор просить вибачення, що не наводить строгої аргументації всіх цих тверджень.)

Перевірка № 3. Спробуємо інше спостереження. Для приємних маршрутів сума модулів усіх кутів повороту становить рівно 2π; для неприємних перевищує 2π. (Автор знову просить вибачення за відсутність строгої аргументації.) Порахувати ці модулі кутів повороту можна як

$$|\alpha_i| = \arccos \frac{\overrightarrow{A_{i-1}A_i} \cdot \overrightarrow{A_iA_{i+1}}}{|A_{i-1}A_i| \cdot |A_iA_{i+1}|} \quad \begin{array}{l}\text{(з очевидними змінами}\\ \text{для } i{=}1 \text{ та } i{=}N\text{);}\end{array} \tag{17}$$

залишається тільки перевірити, чи рівна 2π сума всіх таких $|\alpha_i|$.

Суто теоретично, ця перевірка № 3 самодостатня, досить було б перевіряти лише її, без №№ 1–2. Але, на жаль, фактично так не виходить: arccos (незалежно від того, чи він є готовий бібліотечний, чи його ще треба виражати як $\arccos(x) = \begin{cases} \operatorname{arctg} \frac{\sqrt{1-x^2}}{x}, & 0 < x \leqslant 1 \\ \pi + \operatorname{arctg} \frac{\sqrt{1-x^2}}{x}, & -1 \leqslant x < 0 \\ \frac{\pi}{2}, & x = 0 \end{cases}$) дає вельми велику похибку, тож результатом порівняння «сума $= 2\pi$» буде практично завжди `false` (навіть там, де повинен бути `true`). Це стандартна проблема похибок, і вона має більш-менш стандартне рішення — слід замінити перевірку рівності на перевірку деякої нерівності у стилі «$1{,}999\pi \leqslant$ сума $\leqslant 2{,}001\pi$». Але біда в тому, що після цього результат іноді стає `true` там, де повинен бути `false` (маршрут має один чи кілька «зубців», подібних до 2-го прикладу з умови, але значно менших).

А от якщо робити і цю перевірку № 3, і перевірку № 1 — отримуємо правильний спосіб, достатньо стійкий до похибок. (Автор знову просить вибачення за відсутність строгої аргументації.)

Злиття та два вказівники

1. Теоретичний матеріал

1.1. Злиття

Злиття (рос. «*слияние*», англ. «*merge*») часто розглядають як складову рекурсивного сортування злиттям; ми поставимо на перше місце злиття саме по собі; з рекурсивним сортуванням злиттям теж наполегливо радимо ознайомитися, але за іншими джерелами.

Нехай `a` та `b` — дві відсортовані послідовності, наприклад, чисел. (Те, що це чи́сла — не істотно, може бути й інший тип. Те, що відсортовані — істотно: для невідсортованих вхідних даних злиття не працює.) Результатом злиття буде відсортована послідовність, куди входять усі елементи `a` та `b`.

Нехай вхідні дані задані у відсортованих `vector`-ах `a` та `b`, результат треба повернути як `vector`–результат функції. (Де `vector` — це така сучасна версія масиву, яка сама знає свій розмір (метод `size()`) і вміє додавати елемент собі в кінець (метод `push_back(...)`). Надалі називатимемо `vector` масивом, бо відмінності суто технічні.)

Результат (`res`) спочатку порожній. Елемент `a[0]` мінімальний серед усіх елементів масиву `a`, елемент `b[0]` — серед масиву `b`. Отже, менший із цих двох елементів є мінімальним серед узагалі всіх елементів `a` та `b`, тобто його і треба розмістити у масив `res` як 0-й (мінімальний) елемент об'єднання. Цей елемент стає розглянутим, і до нього вже не треба повертатися. Тож можна продовжити робити те саме для решти елементів. А щоб перейти до них, зсуваємо поточну позицію (індекс) у тому з масивів, звідки взятий цей мінімальний елемент, `i` далі проводимо аналогічні порівняння `a[i]` та `b[j]`.

Важливо усвідомити, що при злитті значення `i` (індекс поточної позиції масиву `a`) та `j` (індекс поточної позиції масиву `b`) рухаються окремо. Цілком можливі ситуації, коли багато разів поспіль зсувався `i` при нерухомому `j`, або навпаки, або вони можуть зсуватися почергово — це істотно залежить від вхідних даних (елементів масивів).

Завершується злиття, коли вже нема чого зливати, бо дійшли до кінця. Зазвичай, якийсь один зі вхідних масивів закінчується, коли в іншому ще є необроблені елементи (можливо, багато). Тому́ «основну части-

ну» злиття треба робити, поки ще є вхідні дані в обох масивах; після «основної частини» треба ще переписати «хвіст» недообробленого масиву (якщо є).

1.1.1. Щодо включення чи виключення повторів при злитті

Можлива ситуація, коли у масивах є однакові елементи `a[i]` = `b[j]`; що робити в таких випадках, істотно залежить від того, який ми хочемо отримати результат. Якщо повтори значень допускаються і хочемо, щоб усі ці повтори були включені у результат, то слід просто виділити два випадки, наприклад, «`a[i]<=b[j]`» та «інакше»:

```
template<class T> vector<T> merge_union(const vector<T> &a, const vector<T> &b)
{
  vector<T> res;//створюємо порожній масив res, щоб будувати у ньому результат
  size_t i = 0, j = 0;//i буде номером поточного ел-та масиву a, j -- масиву b
  // встановлюємо поточні позиції обох векторів на початок
  while(i < a.size() && j < b.size())
  { // поки одночасно в обох масивах ще є необроблені елементи
    if(a[i] <= b[j])
    {//записуємо у результат поточне значення з масиву a, бо воно менше-рівне
      res.push_back(a[i]);
      i++; // зсуваємо позицію (робимо поточним наступний елемент)
    } else // якщо if(b[j] < a[i])
    { // записуємо у результат поточне значення з масиву b, бо воно менше
      res.push_back(b[j]);
      j++; // зсуваємо позицію (робимо поточним наступний елемент)
    }
  }
  while(i < a.size())//якщо у масиві b ел-ти вже скінчилися, а в масиві a ще ні,
    res.push_back(a[i++]); // дописуємо <<хвіст>> масиву a у результат
  while(j < b.size()) // симетрично, якщо закінчилися лише в масиві a, то
    res.push_back(b[j++]); // дописуємо у результат <<хвіст>> масиву b
  return res; // повертаємо res як результат функції
}
```

Са́ме така версія злиття використовується, зокрема, у сортуванні.

(Ні `vector`-и, ні `template`-и не є обов'язковими засобами реалізації злиття. Цілком корисно й доступно написати злиття самостійно, на основі лише раніше наведених словесних описів та користуючись добре відомими засобами (`vector`-и можна замінити простими масивами, можна без `template`-ів указати конкретний тип, наприклад, масив `int`-ів). Просто `vector` дозволяє зручно

додавати елемент у кінець (`push_back`), а `template` дозволяє зробити зразу для всіх типів, а не лише для `int`...)

Нерідко розглядають іншу версію злиття, де гарантовано, що кожна з вхідних послідовностей не містить повторень (упорядкована за зростанням, а не неспаданням), і таку саму впорядкованість за зростанням, а не неспаданням, хочуть мати також і в результуючій послідовності (у випадку, коли одне й те саме число є в обох вхідних послідовностях, воно повинно потрапити у результат, але лише один раз). Найлегше зробити це так: при порівнянні поточних елементів `a[i]` та `b[j]` розрізняти не два випадки, а три: `a[i]` $<$ `b[j]`, `a[i]` $>$ `b[j]`, `a[i]` $=$ `b[j]`.

```
template<class T> vector<T> merge_union(const vector<T> &a, const vector<T> &b)
{
  vector<T> res;//створюємо порожній масив res, щоб будувати у ньому результат
  size_t i = 0, j = 0;//i буде номером поточного ел-та масиву a, j -- масиву b
  // встановлюємо поточні позиції обох векторів на початок
  while(i < a.size() && j < b.size())
  { // поки одночасно в обох масивах ще є необроблені елементи
    if(a[i] < b[j])
    { // записуємо у результат поточне значення з масиву a, бо воно менше
      res.push_back(a[i]);
      i++; // зсуваємо позицію (робимо поточним наступний елемент)
    } else if(b[j] < a[i])
    { // записуємо у результат поточне значення з масиву b, бо воно менше
      res.push_back(b[j]);
      j++; // зсуваємо позицію (робимо поточним наступний елемент)
    } else { // a[i]==b[j], тобто елемент присутній в обох множинах a і b.
      res.push_back(a[i]); // записуємо його у результат (один раз),
      i++, j++; // а до наступного значення переходимо в ОБОХ масивах
    }
  }
  while(i < a.size())//якщо у масиві b ел-ти вже скінчилися, а в масиві a ще ні,
    res.push_back(a[i++]); // дописуємо <<хвіст>> масиву a у результат
  while(j < b.size()) // симетрично, якщо закінчилися лише в масиві a, то
    res.push_back(b[j++]); // дописуємо у результат <<хвіст>> масиву b
  return res; // повертаємо res як результат функції
}
```

Приклад об'єднання злиттям $\{2,3,4,6,7\}$ та $\{1,6,8,9,12\}$:

Вхід 1	Вхід 2	Результат	Примітка
2 3 4 6 7	1 | 6 8 9 12	1	$2 > 1$, вибираємо 1
2 | 3 4 6 7	1 | 6 8 9 12	1 2	$2 < 6$, вибираємо 2
2 3 | 4 6 7	1 | 6 8 9 12	1 2 3	$3 < 6$, вибираємо 3

Вхід 1	Вхід 2	Результат	Примітка
2 3 [4] 6 7	1 ▏ 6 8 9 12	1 2 3 [4]	$4 < 6$, вибираємо 4
2 3 4 [6] 7	1 [6] 8 9 12	1 2 3 4 [6]	$6 = 6$, беремо спільне значення 6 (один раз) та зсуваємо обидві поточні позиції
2 3 4 6 [7]	1 6 ▏ 8 9 12	1 2 3 4 6 [7]	$7 < 8$, вибираємо 7
Основний етап злиття завершено, бо закінчилася одна з послідовностей. Розпочинаємо дописування «хвоста».			
2 3 4 6 7 ▏	1 6 [8] 9 12	1 2 3 4 6 7 [8]	дописуємо «хвіст»
2 3 4 6 7 ▏	1 6 8 [9] 12	1 2 3 4 6 7 8 [9]	дописуємо «хвіст»
2 3 4 6 7 ▏	1 6 8 9 [12]	1 2 3 4 6 7 8 9 [12]	дописуємо «хвіст»

1.2. «Два вказівники»

Перш за все, «два вказівники» — назва більш-менш відомого прийому, який точніше було б називати «дві позиції», бо в ньому рідко коли йдеться про вказівники у технічному значенні цього сло́ва (як змінні, що зберігають адреси в пам'яті). Але історично склалася назва «два вказівники».

Як правило, ці «два вказівники» є двома позиціями в деякій послідовності (часто індексами двох елементів у одновимірному масиві). Цей прийом і ця назва з'являються тоді, коли можна підтримувати ці пари так, що деякі неочевидні переходи від однієї пари до іншої дають змогу розв'язати задачу, яка на перший погляд ніби потребу́є розгляду всіх можливих пар елементів.

1.2.1. Задача «Сума»

Розглянемо задачу: *«Є гарантовано відсортований за строгим зростанням масив* `int A[n]`*. («Строге» зростання означає, що кожен наступний елемент строго більший за попередні, тобто не буває різних елементів з однаковим значенням.) Крім нього, задане число* `SUM_NEED`*. Скільки є різних способів задати це число як суму рівно двох значень із масиву? Допускається (якщо так виходить потрібна сума), щоб це був двічі взятий один елемент. Способи* `a[i]+a[k]` *та* `a[k]+a[i]` *(при* $i \neq k$*) вважати різними».*

Зразок найпростішого розв'язку:

```
int res = 0;
for(int i=0; i<n; i++)
  for(int j=0; j<n; j++)
    if(A[i]+A[j] == SUM_NEED)
      res++;
```

$\Theta(1)$, $\Theta(n)$, $\Theta(n\hat{}2)$

Перепишемо цей простий підхід, розвернувши вкладений цикл у зустрічному напрямі та переробивши його з `for` на `while`:

```
int res = 0;
for(int i=0; i<n; i++) {
  int j = n-1;
  while(j>=0) {
    if(A[i]+A[j]==SUM_NEED)
      res++;
    j--;
  }
}
```

Потім зазначимо такі два факти: 1) якщо масив відсортований і нам треба `A[i]+A[j]==SUM_NEED`, то зменшувати `j` (при незмінному `i`) доцільно лише поки $A[i] + A[j] > SUM\_NEED$; отже, `while(j>=0)` можна змінити на `while(j>=0 && A[i]+A[j]>SUM_NEED)`; 2) якщо при деякому $\texttt{i} = i^\star$ умова `A[i]+A[j]<=SUM_NEED` була досягнута при $\texttt{j} = j^\star$ (а при $\texttt{j} = j^\star + 1$ ще було `A[i]+A[`$j^\star+1$`]>SUM_NEED`), то при наступному $\texttt{i} = i^\star + 1$ можна починати з цього самого $\texttt{j} = j^\star$, а не заново з `j=n-1`.

Отримуємо:

```
int res = 0;
int j = n-1;
for(int i=0; i<n; i++) {
  while(j>=0 && A[i]+A[j] > SUM_NEED)
    j--;
  if(A[i]+A[j]==SUM_NEED)
    res++;
}
```

Аналізуючи цей алгоритм так само, як попередній, отримаємо ту саму загальну кількість дій $O(n^2)$, тож поки що не ясно, чим це краще. Але цей алгоритм можна проаналізувати трохи інакше (див. коментарі):

```
int res = 0, j = n-1;
for(int i=0; i<n; i++) {
  while(j>=0 && A[i]+A[j] > SUM_NEED)
    j--;  // цей while СУМАРНО ПО ВСІМ ІТЕРАЦІЯМ for-а
// займе O(n) часу, бо j лише зменшується від n-1
// не далі як до 0. Такі міркування називають
  if(j>=0 && A[i]+A[j]==SUM_NEED) // груповим аналізом
    res++;
}
```

Тобто, насправді тут $\Theta(n)$, просто щоб це показати, треба вжити так званий *груповий аналіз*: вкладений цикл аналізується *сумарно за всіма* ітераціями зовнішнього циклу.

2. Задачі основного дня (15.10.2017)

Цей комплект задач доступний для on-line перевірки на сайті `ejudge.ckipo.edu.ua`, змагання № 61.

Задача A. «Операції над множинами»

Як відомо, множина — це сукупність елементів, що розглядається (вся сукупність) як єдине ціле. Зазвичай вважають, що множина не може містити один і той самий елемент кілька разів (лише або містить, або ні). Взагалі кажучи, елементи можуть бути різної природи, але у цій задачі елементами будуть лише цілі невід'ємні числа. Для множин відомі деякі стандартні операції. Ми розглянемо лише три найстандартніші:

Об'єднання (математичне позначення «$\cup$», у нас позначається `UNION`) двох множин A та B — це множина усіх елементів, що належать хоча б одній із множин A або B. Якщо елемент належить обом множинам, він усе одно враховується один раз.

Перетин (математичне позначення «$\cap$», у нас позначається `INTERSECTION`) двох множин A та B — це множина усіх елементів, що належать обом множинам A та B одночасно.

Різниця (математичне позначення « $\setminus$ » (зворотня коса риска), у нас позначається `DIFFERENCE`) двох множин A та B — це множина усіх елементів, що належать множині A, але не належать B. Ця операція, єдина з трьох зазначених, несиметрична (не комутативна).

Напишіть програму, яка виконуватиме ці операції над множинами цілих невід'ємних чисел, поданих у вигляді монотонно зростаючих послідовностей, формуючи результат теж у вигляді монотонно зростаючої послідовності.

Вхідні дані

Вхідні дані завжди містять рівно 5 рядків. 1-й: одне з трьох слів `UNION`, `INTERSECTION` або `DIFFERENCE`. 2-й: єдине ціле число N ($1 \leqslant N \leqslant 123456$), що задає кількість елементів множини A. 3-й: послідовність із рівно N розділених одинарними пробілами чисел-елементів множини A; $0 \leqslant a_1 < a_2 < \ldots < a_N \leqslant 10^9$. 4-й та 5-й рядки задають множину B у такому ж форматі, як 2-й і 3-й множину A.

Результати

Виведіть у один рядок, розділяючи пробілами, усі елементи множини-відповіді. Рядок завершити символом переведення рядка.

Якщо результат не містить жодного елемента (наприклад, дія `INTERSECTION`, а A та B не мають спільних елементів), виведення повинно не містити жодного видимого символа, але містити переведення рядка.

Приклади

Вхідні дані	Результати

`UNION`
`3`
`2 3 5`
`3`
`1 2 4`
	`1 2 3 4 5`

`INTERSECTION`
`3`
`2 3 5`
`3`
`1 2 4`
	`2`

`DIFFERENCE`
`3`
`2 3 5`
`3`
`1 2 4`
	`3 5`

Розбір

Суть, переважно, розібрана у розд. 1.1.1. Реалізації «$\cap$», «$\setminus$» відрізняються від «$\cup$» головним чином тим, у яких гілках потрібно заносити поточний елемент до результуючої множини, а у яких не треба:

	a[i]<b[j]	a[i]>b[j]	a[i]==b[j]	дописування «хвоста» a	дописування «хвоста» b
$A \cup B$ UNION	так	так	так	так	так
$A \cap B$ INTERSECTION	ні	ні	так	ні	ні
$A \setminus B$ DIFFERENCE	так	ні	ні	так	ні

Задача B. «Всюдисущі чи́сла»

Дано прямокутну таблицю $N \times M$ чисел. Гарантовано, що у кожному окремо взятому рядку всі чи́сла різні й монотонно зростають.

Напишіть програму, яка шукатиме перелік (також у порядку зростання) всіх тих чисел, які зустрічаються в усіх N рядках.

Вхідні дані

У першому рядку задано два числá N та M. Далі йдуть N рядків, кожен із яких містить рівно M розділених пробілами чисел (гарантовано у порядку зростання).

Результати

Програма має вивести в один рядок через пробіли у порядку зростання всі ті чи́сла, які зустрілися абсолютно в усіх рядках. Кількість чисел виводити не треба. Після виведення всіх чисел потрібно зробити одне переведення рядка. Якщо нема жодного числá, що зустрілося в усіх рядках, виведення повинно не містити жодного видимого символа, але містити переведення рядка.

Приклад

Вхідні дані	Результати
4 5 6 8 10 13 19 8 9 13 16 19 6 8 12 13 15 3 8 13 17 19	8 13

Оцінювання

У 20% тестів $3 \leqslant N, M \leqslant 20$, значення чисел від 0 до 100.
У ще 20% $3 \leqslant N, M \leqslant 20$, значення чисел від -10^9 до $+10^9$.
У ще 20% $1000 \leqslant N, M \leqslant 1234$, значення від 0 до 12345.
У решті 20% $1000 \leqslant N, M \leqslant 1234$, значення від -10^9 до $+10^9$.

Здавати потрібно одну програму, а не чотири; різні обмеження вказані, щоб пояснити, скільки балів можна отримати, розв'язавши задачу не повністю. (Враховуючи особливості саме цього туру, де досить великий штраф за хоча б один не пройдений тест, відсотки балів приблизно дорівнюють трьом чвертям згаданих відсотків тестів.)

Розбір

Пропонується побудувати перетин (див. попередню задачу) всіх рядків (він і є відповіддю). Будувати пропонується послідовно, спочатку 1-й рядок із 2-м, потім їх результат із 3-м, і так до кінця.

Задача C. «Школярі з хмарочосів»

У школі вчаться діти, які проживають у двох будинках-хмарочосах, розташованих поруч зі школою. Для того, щоб дійти до школи від 1-го хмарочоса, потрібно t_1 часу, а від 2-го потрібно t_2 часу. У 1-му хмарочосі живуть N_1 школярів, у 2-му N_2. Про кожного школяра відомий час, коли він виходить із під'їзду.

Напишіть програму, яка з'ясовуватиме, у якому порядку вони приходитимуть до школи.

Вхідні дані

Складаються з рівно шести рядків. Перший рядок містить єдине число — час, потрібний щоб дійти від 1-го хмарочоса до школи. Другий рядок містить єдине число N_1 — кількість школярів, які проживають у 1-му хмарочосі. У третьому рядку через пробіли записані (гарантовано впорядковані за строгим зростанням) моменти часу, коли школярі виходять із під'їзду. Рядки з четвертого по шостий описують, у такому са-

мому форматі, учнів 2-го хмарочоса. Кількості учнів N_1 та N_2 можуть бути як однаковими, так і різними, кожна не менша 1 і не більша 98765. Усі значення часу є цілими числами у проміжку від 1 до 12345678.

Результати

Потрібно вивести перелік школярів у тому порядку, як вони приходять у школу. Дані кожного школяра мають бути виведені в окремому рядку, кожен такий рядок мусить мати вигляд: час, коли учень приходить до школи; номер хмарочоса, де він живе; номер, яким по порядку він виходить із під'їзду свого хмарочоса. Якщо різні учні приходять до школи одночасно (це можливо лише для учнів із різних хмарочосів), слід виводити спочатку дані про учня з 1-го хмарочоса, потім дані про учня з 2-го.

Приклад

Вхідні дані	Результати
10 3 0 4 7 15 2 0 3	10 1 1 14 1 2 15 2 1 17 1 3 18 2 2

Розбір

Пропонується робити «зсунуте злиття», тобто щоразу порівнювати не просто `a[i]` з `b[j]`, а `a[i]+t1` з `b[j]+t2`:

```
while (i<=n)and(j<=m) do begin
  if a[i]+t1 <= b[j]+t2 then begin
    writeln(a[i]+t1, ' 1 ', i);
    inc(i);
  end else begin
    writeln(b[j]+t2, ' 2 ', j);
    inc(j);
  end;
end;
```

```
while i<=n do begin // <<хвіст>> a
  writeln(a[i]+t1, ' 1 ', i);
  inc(i);
end;
while j<=m do begin // <<хвіст>> b
  writeln(b[j]+t2, ' 2 ', j);
  inc(j);
end;
```

Задача D. «Відстань між мінімумом та максимумом»

Напишіть програму, яка за заданим масивом цілих чисел знайде місце мінімального її елемента, максимального її елемента та відстань (кількість проміжних елементів) між ними. У випадку, якщо масив містить однакові мінімальні та/або однакові максимальні елементи на різних позиціях, слід вибрати такий із мінімальних і такий із максимальних елементів, щоб шукана відстань виявилася мінімальною (див. також приклади 3 та 4).

Вхідні дані

Перший рядок містить єдине число N ($2 \leqslant N \leqslant 10^6$) — кількість елементів масиву. Другий рядок містить (розділені пробілами) числа-елементи масиву; значення цих чисел-елементів цілі, від 1 до 10^6.

Результати

Виведіть єдине ціле число — мінімально можливу відстань (кількість проміжних елементів) між якимсь із мінімальних і якимсь із максимальних елементів.

Приклади

Вхідні дані	Результати	Примітки
8 3 9 7 5 2 8 6 4	2	Між елементами 2 та 9 два проміжні елементи 5 та 7
8 4 3 2 1 8 7 6 5	0	Між елементами 1 та 8 немає проміжних елементів

Вхідні дані	Результати	Примітки
8 3 1 4 1 5 9 2 6	1	Максимальний елемент (дев'ятка) єдиний; найближчою до єдиної дев'ятки є остання з мінімальних елементів (одиниць); між ними один проміжний елемент 5
8 1 1 1 1 1 1 1 1	0	Оскільки всі елементи однакові, кожен із них мінімальний і максимальний; між елементом і ним же самим немає проміжних

Оцінювання

У 20% тестів масив містить єдине мінімальне та єдине максимальне значення; у 80% і мінімальне, і максимальне повторюються. Розподіл тестів за розмірами такий: 20% припадає на тести, де $2 \leqslant N \leqslant 100$; 30% — на тести, де $4000 < N \leqslant 10^4$; 20% — на тести, де $5 \cdot 10^4 < N \leqslant 10^5$; 30% — на тести, де $7 \cdot 10^5 \leqslant N \leqslant 10^6$. Писати різні програми для різних випадків не треба; розподіли вказані, щоб показати, скільки приблизно балів можна отримати, розв'язавши задачу не повністю. (Враховуючи особливості саме цього туру, де досить великий штраф за хоча б один не пройдений тест, відсотки балів приблизно дорівнюють трьом чвертям згаданих відсотків тестів.)

Розбір

Пропонується сформувати два масиви: в одному всі індекси, де зустрічається мінімальне значення, в іншому — всі індекси, де зустрічається максимальне значення. Потім пройти по цих масивах, використовуючи злиття, але не формуючи послідовність-результат, а шукаючи мінімум серед усіх значень `abs(a[i]-b[j])` (потім ще 1 відняти наприкінці, якщо тільки це не буде ситуація, коли відповідь і так 0).

```
// у vector<int> MI складені всі індекси,
// де зустрічалося мінімальне значення,
// у vector<int> MA -- всі індекси, де максимальне
size_t i=0, j=0;
int res = abs(MI[0] - MA[0]);
```

```
while(i < MI.size() && j < MA.size()) {
  res = min(res, abs(MI[i] - MA[j]));
  if(MI[i] <= MA[j]) {
    i++;
  } else {
    j++;
  }
}
res = max(0, res-1);
```

Задача E. «Сума»

Є *гарантовано відсортована* за строгим зростанням послідовність з n цілих чисел і число SUM_NEED. («Строге» зростання означає, що кожен наступний елемент строго більший за попередні, тобто не буває різних елементів з однаковим значенням.) Скільки є різних способів задати це число як суму $a[i] + a[k]$? (Тобто, як суму з рівно двох доданків; допускається, якщо так виходить потрібна сума, щоб це був двічі взятий один елемент; способи $a[i] + a[k]$ та $a[k] + a[i]$ (при $i \neq k$) вважаються різними.)

Вхідні дані

Перший рядок містить кількість елементів послідовності n ($1 \leqslant n \leqslant \leqslant 123456$). Другий рядок містить n чисел — самі елементи. Третій рядок містить потрібну суму SUM_NEED.

Результати

Єдине число — кількість способів отримати SUM_NEED як $a[i] + a[k]$.

Приклад

Вхідні дані	Результати
7 2 3 5 7 11 13 17 22	3

Примітка

Цими трьома способами є: 1) $5 + 17$; 2) $11 + 11$; 3) $17 + 5$.

Задача повністю розібрана у розд. 1.2.1.

Задача F. «Хвіртка у паркані»

Пан Дивак вирішив оновити паркан, що відділяє його подвір'я від вулиці. Він вже закопав N стовпчиків, а потім згадав, що у паркані бажано б залишити хвіртку, шириною щонайменше W. Тепер йому, мабуть, доведеться викопувати деякі з вкопаних стовпчиків.

Щоб робота не була даремною, слід викопати якнайменше стовпчиків. Допоможіть панові Диваку визначити, скільки стовпчиків доведеться викопати. Після викопування стовпчиків мусить існувати проміжок (між двома залишеними стовпчиками, або між залишеним стовпчиком і одним із кінців ділянки, або між кінцями ділянки) ширини $\geqslant W$.

Вхідні дані

Перший рядок містить два цілих числа N та W — кількість вкопаних стовпчиків і мінімально необхідну ширину проміжку для хвіртки відповідно. Гарантується, що $0 \leqslant N \leqslant 100000$ і $0 \leqslant W \leqslant 10^9$. Вважатимемо, що уздовж межі подвір'я введено вісь координат. У другому рядку вхідного файла вказано два числа L та R ($L < R$) — координати лівого і правого кінців межі подвір'я. Далі йде третій рядок, що містить N чисел — координати вкопаних стовпчиків. Усі координати (включаючи L та R) — різні цілі числа, що не перевищують за модулем (абсолютною величиною) 10^9. Гарантується, що всі стовпчики вкопані між лівим і правим кінцями.

Результати

Слід вивести єдине число у єдиному рядку — мінімальну кількість стовпчиків, які треба викопати. Якщо розв'язку не існує, то виведіть замість кількості число «-1» (без лапок).

Приклади

Вхідні дані	Результати
3 2 2 6 3 4 5	1
3 2 1 6 4 3 5	0
3 5 1 7 5 3 4	3

Примітка

Гарантовано, що хоча б у половині тестів координати стовпчиків відсортовані за зростанням.

Розбір

Окремо слід розглянути й відкинути украй нелогічний, але формально дозволений випадок $R - L < W$ (ширина хвіртки більша за відстань між межами подвір'я). Саме цей випадок є єдиною ситуацією, коли слід виводити відповідь «-1».

Координати стовпчиків слід відсортувати. (Причому, враховуючи обмеження на N, ефективним алгоритмом. Якщо мова програмування містить готове бібліотечне сортування, як-то функція `sort` бібліотеки `algorithm` мови C++, можна ним і скористатися. Якщо писати сортування самому, це може бути, наприклад, quickSort чи пірамідальне сортування; або це може бути сортування, основане на рекурсивному поділі та злитті.) Координати кінців подвір'я L та R варто включити у початок і кінець того самого масиву, але пам'ятати, що вони мають особливий смисл.

Далі слід застосувати прийом «два вказівники», де «вказівники» рухаються не назустріч, а в одному й тому ж напрямку, підтримуючи властивість «відстань між двома поточними стовпчиками (чи стовпчиком і межею подвір'я) якнайближча згори до W».

Розглянемо детальніше, що все це означає. Нехай координати стовпчиків і кінців подвір'я вже складені в один масив `data` та відсортовані, значення `N` вже збільшено на 2 (бо тепер масив `data` містить не лише стовпчики, а й ме́жі подвір'я). Тоді фрагмент

```
i:=1;  j:=2;
while data[j] - data[i] < W do
  inc(j);
best_ans:=j-2;
```

знайде, перед яким стовпчиком (чи кінцем подвір'я) завершиться прохід шириною W, якщо відкласти його від лівої межі́ подвір'я. (Це справді так, бо стовпчики та ме́жі відсортовані, і для всіх пропущених відстань була менша W, а зупинилися якраз там, де стало $\geqslant W$, тобто досить місця для хвіртки. Виходу за межі масиву не буде, бо випадок $R - L < W$ вже розглянули і відкинули.) Отже, якщо ставити хвіртку саме тут, доведеться викопати `j-2` стовпчики. Запам'ятаємо це як можливу відповідь (а потім, можливо, покращимо — за стандартним шкільним алгоритмом пошуку мінімуму).

Для кожного подальшого `i` (стовпчика, відразу після якого пробуватимемо починати хвіртку) немає потреби розглядати ті `j`, котрі вже були відкинуті, а можна продовжити якраз із того місця, де зупинилися при попередньому `i`. Усе разом може набути такого, наприклад, вигляду:

```
while (j<=N) do begin
  if j-i-1 < best_ans then
    best_ans := j-i-1;
  inc(i);
  while (j<=N) and (data[j] - data[i] < W) do
    inc(j);
end;
```

Задача G. «Сума та різниця Мінковського»

Наведемо два різні формулювання однієї й тієї самої задачі.

Сума Мінковського двох фігур на координатній площині A, B — це фігура на координатній площині, якій належать ті й тільки ті точки, координати яких (x, y) можна подати як $x = x_1 + x_2$, $y = y_1 + y_2$, де

(x_1, y_1) — довільна з точок, що належать фігурі A, а (x_2, y_2) — довільна з точок, що належать фігурі B. Ця операція можлива для будь-яких фігур A, B.

Різниця Мінковського двох фігур на координатній площині A, B — це така фігура C на координатній площині, що фігура A є сумою Мінковського фігур B та C. Ця операція означена лише для деяких фігур A, B (а для решти пар A, B підібрати таку C неможливо).

Напишіть програму, яка читатиме описи фігур A, B і знаходитиме суму та різницю Мінковського цих фігур. На вхід гарантовано подаватимуться лише фігури, котрі є опуклими многокутниками, причому одна зі сторін такого многокутника горизонтальна, ліва вершина цієї сторони має координати $(0, 0)$, а решта сторін знаходяться у півплощині з додатними y.

Напишіть програму, яка вмітиме знаходити:

1) суму Мінковського;
2) різницю Мінковського.

Вхідні дані

Спочатку 1 або 2 на позначення того, яку задачу потрібно розв'язувати («1» — суму, «2» — різницю), потім описи опуклих многокутників A та B у такому форматі: спочатку число N ($3 \leqslant N \leqslant 1000$) — кількість вершин у многокутнику, а далі N груп по 2 цілих числá x_i та y_i — координати вершин. Система координат завжди вибирається так, що перша вершина має координати $(0; 0)$, остання має координати $(0; x_n)$, де $x_n > 0$. Усі вхідні координати — цілі чи́сла з проміжку від 0 до 10^6.

Усі чи́сла записано в один рядок і розділено пропусками (пробілами).

Результати

Якщо розв'язується задача «2» і різниці для вказаних многокутників не існує, то вивести єдине число 0. Інакше вивести результат суми чи різниці Мінковського в тому ж форматі, що для вхідних даних. Гарантовано, що використовуються лише такі вхідні дані, що якщо відповідь існує, то всі її координати є цілочисельними.

На станції Глупов-Товарний використовуються підйомні крани спеціальної конструкції «Мостовий-Глуповський». Гак такого крана під-

вішений до кількох блоків, що їздять по рейці, розміщеній горизонтально (на певній висоті). Завдяки цьому гак можна переміщати в будь-яку точку частини площини, обмежену многокутником спеціального вигляду: верхня сторона многокутника збігається з рейкою крана, обидва внутрішні кути многокутника при цій стороні гострі, решта вершин многокутника розміщені довільним чином, але так, що многокутник є опуклим. Крім того, станція має в розпорядженні пристрій, що дає змогу комбінувати дію двох кранів такого типу: простір досяжності гака скомбінованого механізму точно такий, ніби рейку другого крана підвісили (зі збереженням горизонтальності) на гак першого.

Напишіть програму, яка вмітиме виконувати такі дві дії:

1. За заданими областями досяжностей двох кранів знаходити область досяжності їхньої комбінації.
2. За заданими областю досяжності одного крана та потрібною областю досяжності з'ясовувати, який потрібно взяти другий кран, щоб комбінація першого та другого кранів у точності співпадала з потрібною областю досяжності (або з'ясовувала, що це неможливо).

Вхідні дані

Спочатку `1` або `2` (на позначення того, яку задачу потрібно розв'язувати), потім ідуть дві області. Якщо розв'язується задача «1», то це області досяжності першого та другого кранів, якщо «2», то спочатку потрібна область досяжності, а потім область досяжності першого крана. Усі області досяжності задаються у такому форматі: спочатку число N ($3 \leqslant N \leqslant 1000$) — кількість вершин у многокутнику, а далі N груп по 2 цілих числá x_i та y_i — координати вершин цієї області в порядку зростання x-координати. Система координат завжди вибирається так, що перша вершина має координати (0; 0), вісь y напрямлена згори донизу. Всі вхідні координати — цілі чи́сла, що не перевищують за модулем (абсолютною величиною) 10^6.

Усі чи́сла записано в один рядок і розділено пропусками (пробілами).

Результати

Якщо розв'язується задача «2» і підібрати другий кран неможливо, то вивести на екран єдине число 0. Інакше вивести побудовану область досяжності (в тому ж форматі, що для вхідних даних). Гарантовано,

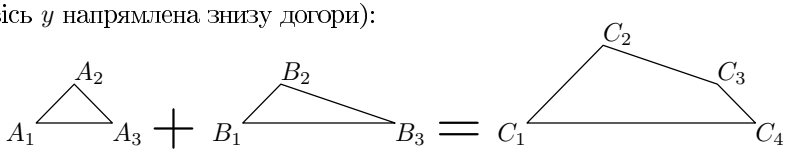
що використовуються лише такі вхідні дані, що якщо відповідь існує, то всі її координати є цілочисельними.

Приклади

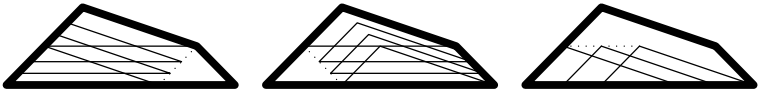
Вхідні дані	Результати
2 5 0 0 10 40 20 50 30 40 40 0 3 0 0 10 10 20 0	3 0 0 10 40 20 0
2 3 0 0 10 10 20 0 3 0 0 10 10 40 0	0
1 3 0 0 10 10 20 0 3 0 0 10 10 40 0	4 0 0 20 20 50 10 60 0

Примітка

Пояснимо останній із наведених прикладів вхідних даних і результатів детальніше. Самі доданки та сума мають такий вигляд (за умови, що вісь y напрямлена знизу догори):



Цей результат може бути утворений, наприклад, унаслідок «ковзань» другого многокутника уздовж сторін першого.



(a) (b) (c)

Розбір

Примітка дає дуже багато для розв'язання задачі: з неї видно, що кожна зі сторін многокутника-суми (як вільний вектор) являє собою або одну зі сторін одного з многокутників-доданків, або (у випадку, якщо у многокутниках-доданках є співнапрямлені вектори) суму таких співнапрямлених векторів. Опуклість многокутників-доданків означає, що сторони в порядку обходу є відсортованими за тригонометричним кутом; са́ме це дає можливість застосувати злиття. Схема злиття (для суми) така:

```
int i=0; // номер поточного ребра у многокутнику-доданку A
int j=0; // номер поточного ребра у многокутнику-доданку B
while(i < A.size() && j < B.size()) {
  обчислити кут повороту від ребра A[i] до ребра B[j]
  if(від'ємний, тобто за годинниковою стрілкою) {
    додати у результат вектор A[i];  i++;
  } else if(додатний, тобто проти годинникової стрілки) {
    додати у результат вектор B[j];  j++;
  } else { // нульовий, тобто співнапрямлені
    додати у результат суму векторів A[i] + B[j];
    i++;  j++;
  }
}
```

Зрозумівши хід побудови суми, неважко зрозуміти також і хід побудови різниці:

```
int i=0; // номер поточного ребра у многокутнику A (зменшуваному)
int j=0; // номер поточного ребра у многокутнику B (від'ємнику)
while(i < A.size() && j < B.size()) {
  обчислити кут повороту від ребра A[i] до ребра B[j]
  if(від'ємний, тобто за годинниковою стрілкою) {
    додати у результат вектор A[i];  i++;
  } else if(додатний, тобто проти годинникової стрілки) {
    обірвати цикл, запам'ятавши, що результат усієї дії --
        <<різниці не існує>>
  } else { // нульовий, тобто співнапрямлені
    if(A[i] дорівнює B[j]) {
      i++;  j++;  // нічого не додаючи у результат,
          // бо це із суми якраз забрали від'ємник
    } else if(A[i] строго коротший за B[i]) {
      обірвати цикл, запам'ятавши, що результат усієї дії --
          <<різниці не існує>>
    } else { // A[i] строго довший за B[i]
      додати до результату різницю векторів A[i] - B[j]
      i++;  j++;
    }
  }
}
```

Ірина Скляр, заслужений вчитель України, завідувач кафедри інформатики Київського природничо-наукового ліцею № 145.

Теорія чисел у програмуванні

Короткі відомості з теорії чисел

Теорія чисел – розділ математики, у якому вивчаються властивості чисел. Основний об'єкт вивчення цієї теми – натуральні числа. Основною їх властивістю є подільність, яка і розглядатиметься у поданому матеріалі.

Одні з перших задач, із якими знайомляться в теорії чисел, вимагають уміння розкладати натуральне число на множники. Основними «цеглинками» такого розкладання є так звані *прості числа*.

Означення. Натуральне число називається ***простим***, якщо воно має тільки два дільники – одиницю та саме число. Натуральне число, що має більше двох дільників, називається ***складеним***.

Представлення числа N у вигляді $N = p_1^{a_1} \cdot p_2^{a_2} \cdot \ldots \cdot p_n^{a_n}$, де p_i – прості числа, називається розкладанням числа N на прості множники, або ***канонічним*** розкладанням.

Основна теорема арифметики. Кожне натуральне число однозначно представляється у вигляді добутку простих чисел із точністю до перестановки множників.

Виходячи з цієї теореми, число 1 не є простим, оскільки тоді розкладань буде безліч. Наприклад, число 6 можна розкласти таким чином:

6 = 2*3 = 1*2*3 = 1*2*1*3 і т. д.

Якщо ж вважати, що число 1 не є простим, розкладання буде єдиним:

6 = 2*3.

Для більшості задач, що будуть запропоновані у поданому матеріалі, потрібно визначити, чи є число простим.

Існує кілька методів визначення, чи є задане число простим. Розглянемо спочатку найпростіший повноперебірний. Він полягає у тому, що перебираються *усі можливі* дільники заданого числа. Якщо в процесі перебору буде знайдено хоч одне число, що є дільником заданого, число, що перевіряється, – не є простим.

Для учнів найскладнішим у цьому алгоритмі є визначення усіх можливих дільників. Якщо найменший дільник ще є очевидним – це

двійка (на одиницю перевіряти подільність немає сенсу), то пошук найбільшого вже викликає труднощі. Учні, як правило, пропонують число $\boldsymbol{N}$–**1** (найбільше з чисел, на яке не ділиться N). Однак після нескладного аналізу вони все ж таки озвучують більш прийнятний варіант – $\boldsymbol{N/2}$. Дійсно, кількість зайвих переборів зменшується удвічі. Однак, і цього недостатньо – задача буде розв'язуватися за неприйнятно довгий час із числами порядку 10^9 або більше.

Щоб суттєво зменшити кількість переборів, потрібно згадати властивість складених чисел: якщо число має дільники, відмінні від одиниці, то принаймні один із них буде знаходитися в діапазоні до квадратного кореня з числа. Наприклад, число 90 має дільниками 2, 3, 5 і 6, які є меншими, ніж $\sqrt{90}$ (це далеко не всі дільники), а число 25 – має єдиний дільник 5, який дорівнює $\sqrt{25}$.

Такий очевидний алгоритм легко реалізується в такий спосіб:

```
function isPrime(N:longint):boolean;
var i:longint;
chk:boolean;
begin
if N < 2 thenchk := false
else chk := true;
  for i:=2 to round(sqr(n)) do
     if N mod i = 0 then chk := false;
isPrime := chk;
end;
```

Якщо перевірити потрібно одне єдине число, наведений алгоритм, безумовно, можна використати. Але якщо чисел буде багато, знову може виявитися, що час розв'язання задачі занадто великий. Потрібно знайти інший варіант алгоритму, який суттєво зменшить кількість перевірок. Так, наприклад, для числа 1000000 можна завершувати перевірку вже після ділення на 2 (воно ділиться націло), а запропонований алгоритм виконуватиме ще 999 зайвих перевірок на подільність.

Щоб не робити зайві перевірки, пропонується замінити повноперебірний цикл **for** на цикл **while**, який завершуватиме роботу після першого знайденого числа, на яке ділиться задане (як і раніше, числа, що перевищують корінь квадратний із заданого числа, не розглядаються). Крім того, окремими командами розгалуження опрацьовуються прості числа 2 та 3, щоб потім усі парні числа від-

разу виключити з перевірки, а усі непарні перевіряти на подільність тільки на непарні числа. Функція, що реалізує описаний алгоритм, має такий вигляд:

```
Function isPrime(N:longint):boolean;
var i:longint;
chk:boolean;
begin
if N < 2 then chk := false
else if  N < 4 then chk := true
else if  N mod 2 = 0 then chk := false
else begin
              i := 3;
while (i <= sqrt(N))and(N mod i<>0) do
                  i := i + 2;
if N mod i = 0 then chk := false
else chk := true;
end;
isPrime := chk;
end;
```

Для учнів, що розв'язують задачі мовою програмування C++, можна запропонувати таку функцію перевірки числа на простоту:

```
bool prime(long longn)
{
    if (n<2)return false;
    if (n<4) return true;
    if (n%2==0) return false;
for (inti=3; i<=sqrt(n);i+=2)
if (n%i == 0)return false;
    return true;
}
```

Безумовною перевагою цієї мови програмування є те, що після виконання оператора **return** функція завершує свою роботу. Звідси випливає, що слідкувати за зайвими перевірками не потрібно: у разі знаходження числа, на яке ділиться задане, виконується завершення роботи функції з поверненням значення **false**. У протилежному випадку функція повертає значення **true**.

Нарешті, можна, використовуючи алгоритм «решета Ератосфена», сформувати масив **Primes**, який міститиме:

$$Primes[i] = \begin{cases} 1, \textit{ якщо } i \textit{ – просте число} \\ 0, \textit{ якщо } i \textit{ – складне число} \end{cases}$$

Нагадаємо зазначений алгоритм. Запишемо всі натуральні числа, починаючи з 2 до N. Спочатку з цього ряду видаляються всі числа, що більші за 2 та кратні двом, потім усі числа, більші за 3 та кратні трьом, і так далі.

Вибір чисел, які будуть основою для викреслення, виконуватиметься зліва направо. На кожному кроці береться наступне невикреслене число і викреслюються числа, що йому кратні. Максимальне число, яке ще потрібно використовувати для викреслення, не повинно перевищувати квадратного кореня з найбільш можливого числа ряду.

Функція, що генерує масив, у якому одиницями (або значенням **true**) позначені елементи, індекси яких є простими числами, має вигляд:

```
type Tprimes=array[1..100000000] of  Boolean;
procedure generation_primes(varPrimes:Tprimes;n:longint);
vari,j:longint;
begin
  for i:=2 to n do Primes[i]:=true;
  Primes[1]:=false;
  i:=2;
  while (i<=sqrt(n)) do
  begin
    while not Primes[i] do inc(i);
    for j:=2 to n div i do
      Primes[i*j]:=false;
inc(i);
  end;
end;
```

Наведений алгоритм буде найшвидшим для перевірки великої кількості чисел, що надходять на обробку. Однак він має суттєвий недолік: числа, що перевіряються, не можуть перевищувати за значенням 10^8. Можна замінити масив логічних значень бітовим полем, у якому кожен біт відповідає за своє число. Але це збільшить діапазон чисел, що перевіряються, тільки у 8 разів і також не завжди може задовольнити умову задачі.

Можна також створити константний масив простих чисел. Але такий підхід при значному підвищенні швидкодії ще більше зменшить діапазон чисел, що перевіряються. Для запропонованих далі задач такий підхід не потрібен і тому пропонуємо вам спробувати зробити це самостійно.

Крім перевірки числа на простоту, часто зустрічаються задачі на визначення найбільшого спільного дільника двох чисел. Зі шкільного курсу відомо, що найбільшим спільним дільником для чисел a та b є таке найбільше ціле число, на яке ділиться a та b без залишку. При цьому обидва числа одночасно не можуть дорівнювати 0. У випадку рівності обох чисел нулю домовимося вважати їх НСД, рівним нулю.

Позначимо найбільший спільний дільник як НСД(a,b). Виходячи з визначення, справджуються такі рівності:

НСД(a,b) = НСД(b,a);
НСД$(-a,b)$ = НСД(a,b);
НСД$(a,0)$ = $|a|$;
НСД$(0,0)$ = 0.

Із поняттям найбільшого спільного дільника дуже тісно пов'язане поняття найменшого спільного кратного. Найменшим спільним кратним двох цілих чисел називається таке найменше додатне ціле число, яке ділиться націло на перше та друге число (кратне обом цим числам).

Відомий із курсу математики алгоритм пошуку найбільшого спільного дільника спирається на основну теорему арифметики. Дійсно, запишемо для обох чисел їх канонічний розклад:

$$a = p_1^{a_1} \cdot p_2^{a_2} \cdot \ldots \cdot p_n^{a_n};$$
$$b = p_1^{b_1} \cdot p_2^{b_2} \cdot \ldots \cdot p_n^{b_n}.$$

З нього слідує, що

$$НСД\,(a,b) = p_1^{\min(a_1,b_1)} \cdot p_2^{\min(a_2,b_2)} \cdot \ldots \cdot p_n^{\min(a_n,b_n)}.$$

Наприклад, розглянемо числа 48 і 72. Канонічне розкладання цих чисел має вигляд:

$$48 = 2^4 \cdot 3;$$
$$72 = 2^3 \cdot 3^2.$$

Знаходимо найбільший спільний дільник цих чисел:

$$НСД\,(48,72) = 2^{\min(3,4)} \cdot 3^{\min(1,2)} = 2^3 \cdot 3^1 = 24.$$

Цей відомий із курсу математики алгоритм дуже складно програмується. Набагато простіше запрограмувати інший алгоритм – так званий алгоритм Евкліда. Він полягає у тому, що потрібно від більшого числа віднімати менше, доки числа не стануть рівними. Наприклад, для чисел 48 та 72 результат виконання алгоритму виглядатиме таким чином:

a	b
48	72
48	72 – 48 = 24
48 – 24 = 24	24
24	24

Програмується цей алгоритм дуже просто:

```
while (a<>b) do
begin
   if a>b
   then a:=a-b
   else b:=b-a;
end;
```

Але швидкодія цього алгоритму є незадовільною, особливо коли числа сильно відрізняються за значенням одне від одного, наприклад, 2 та 10000890. Щоб значно пришвидшити алгоритм, проведемо його аналіз. Для цього спробуємо знайти НСД для чисел 6 і 1024. У цьому випадку нам доведеться багаторазово віднімати від числа 1024 число 6, оскільки результат все одно буде більшим, ніж 6. Після виконання віднімання 170 разів число 1024, нарешті, заміниться на 4 і стане меншим, ніж 6. Далі число 6 заміниться на 2 (6 – 4), а потім 4 – на 2 (4 – 2).

Очевидно, що виконуване багаторазове віднімання можна замінити цілочисельним діленням. І, оскільки нас цікавить тільки остаточний результат віднімання, замінюватимемо більше число не на різницю, а на залишок від ділення цього числа на друге. Кількість виконуваних операцій при цьому значно зменшиться. Зверніть увагу, що зміниться умова завершення роботи алгоритму: тепер завершення буде у випадку, коли принаймні одне з чисел стане дорівнювати 0. Для чисел 1024 та 6 виконання алгоритму виглядатиме так:

a	B
1024	6
1024 mod 6 = 4	6
4	6 mod 4 = 2
4 mod 2 = 0	2
2	2

Як бачите, кількість кроків виконання алгоритму стала дорівнювати 3 замість 172 при використанні попереднього підходу.

Функція, яка знаходить найбільший спільний дільник за модифікованим алгоритмом Евкліда, виглядає так:

```
function NSD(a,b:int64):int64;
begin
a:=abs(a);
  b:=abs(b);
while (a<>0)and(b<>0) do
begin
 if a>b
then a:=a mod b
else b:=b mod a;
end;
NSD:=a+b;
end;
```

Зверніть увагу: на початку функції знаходяться модулі чисел, ураховуючи, що найбільший спільний дільник – це додатне число. Результатом алгоритму є сума двох чисел, оскільки одне з них обов'язково перетвориться на 0. Крім того, це не суперечить тому, що для двох нулів найбільший спільний дільник дорівнює 0.

Якщо потрібно знайти найбільший спільний дільник послідовності з N чисел, то спочатку за НСД береться перше число, а потім знаходиться НСД нового числа і вже отриманого НСД. Реалізація описаного алгоритму матиме вигляд:

```
function NSD(a,b:int64):int64;
begin
    a:=abs(a);  b:=abs(b);
while(a<>0) and (b<>0)do
if (a>b)
then a:=a mod b
else b:=b mod a;
    NSD:=a+b;
end;
varN,i:longint; NSD_one,num:int64;
fin,fout:text;
begin
assign(fin,'input.txt');reset(fin);
assign(fout,'output.txt');rewrite(fout);
read(fin,N);
read(fin,NSD_one);
```

```
for i:=2 to N do
begin
read(fin,num);
NSD_one:=NSD(NSD_one,num);
end;
writeln(fout,NSD_one);
close(fin);
close(fout);
end.
```

Мовою C++ розв'язок виглядатиме таким чином:

```
#include <bits/stdc++.h>
int NSD(long long a, longlong b)
{
    a=abs(a);  b=abs(b);
while(a!=0 && b!=0)
if (a>b) a%=b;
else b%=a;
return a+b;
}
using namespace std;
int main()
{
int N;
long long a,nsd;
freopen(«input.txt»,»r»,stdin);
freopen(«output.txt»,»w»,stdout);
cin>> N;
cin>>nsd;
for (int i=1; i<N; i++)
    {
cin>> a;
nsd=NSD(nsd,a);
    }
cout<<nsd<<endl;
return 0;
}
```

Найменше спільне кратне математично визначається аналогічно найбільшому спільному дільнику:

$$HCK(a,b) = p_1^{\max(a_1,b_1)} \cdot p_2^{\max(a_2,b_2)} \cdot \ldots \cdot p_n^{\max(a_n,b_n)}$$

Для тих самих чисел 48 і 72 канонічне розкладання має вигляд:

$$48 = 2^4 \cdot 3$$
$$72 = 2^3 \cdot 3^2$$

А найменше спільне кратне:

$$HCK(48,72) = 2^{\max(3,4)} \cdot 3^{\max(1,2)} = 2^4 \cdot 3^2 = 144.$$

Для програмування знаходження найменшого спільного кратного також складно використовувати наведений алгоритм. Але можна скористатися теоремою зв'язку між НСД та НСК:

$$a \cdot b = NSD(a,b) \cdot NSK(a,b).$$

Із цієї теореми випливає наступне співвідношення:

$$NSK(a,b) = \frac{a \cdot b}{NSD(a,b)}.$$

Щоб не вийти за межі типу, пропонуємо поміняти порядок виконання дій: спочатку виконати ділення, а потім уже множення. Функція, що реалізує заданий алгоритм:

```
function NSK(a,b:int64):int64;
begin
NSK:=a div NSD(a,b) * b;
end;
```

Розбір задач

Задача «Прості числа».

Умова: Дано послідовність натуральних чисел. Напишіть програму, що визначає кількість простих чисел цієї послідовності.

Вхідні дані: З файла зчитується послідовність цілих чисел, кожне з яких знаходиться в окремому рядку. Послідовність завершується числом 0, після якого зчитування потрібно припинити. Усі числа у вхідному файлі не перевищують $2 \cdot 10^9$.

Вихідні дані: Єдине число – кількість простих чисел у заданій послідовності.

Ідея розв'язання задачі. Оскільки зчитування даних здійснюється з файла, потрібні файлові змінні, що дозволяють організувати обмін даними із зовнішнім носієм. Крім того, потрібні змінні для зчитування числа та підрахунку кількості простих чисел у послідовності.

За умовою задачі зчитування повинно виконуватися до першого нуля, тому пропонуємо скористатися циклом із післяумовою, що завершує свою роботу після введення з вхідного потоку нуля.

```
Var num:longint;
counter:longint;
fin,fout:text;
begin
assign(fin,'input.txt'); reset(fin);
assign(fout,'output.txt'); rewrite(fout);
counter := 0;
repeat
read(fin,num);
if isPrime(num) then
inc(counter);
until(num = 0);
writeln(fout,counter);
close(fin);
close(fout);
end.
```

Аналогічно розв'язується задача пошуку перших N простих чисел. Для цього цикл основної програми повинен бути організований таким чином, щоб він завершував свою роботу в разі досягнення лічильником простих чисел значення N. Покажемо, як виглядатиме основна програма при використанні циклу з передумовою.

```
Var N,counter,num:longint;
fin,fout:text;
begin
assign(fin,'input.txt'); reset(fin);
assign(fout,'output.txt'); rewrite(fout);
  read(fin,N);
counter := 0;
num:=1;   //перше натуральне число
while counter<N do
  begin
ifisPrime(num)
then begin
inc(counter);//знайдене чергове просте число
write(fout,num,' ');
    end;
inc(num);    //перехід до наступного натурального числа
end;
close(fin);
close(fout);
end.
```

Дуже схоже розв'язуються задачі пошуку простих чисел, що не перевищують введеного значення N, або у діапазоні від A до B. Різниця тільки в тому, що організовувати підрахунок кількості знайдених чисел не потрібно, а потрібно кожне знайдене просте число виводити у вихідний потік.

У першому випадку організовується повноперебірний цикл **for** від **1** до N:

```
for i:=1 to N do
   if isPrime(i) then write(fout,i);
```

А у другому – від числа A до числа B. Якщо за умовою задачі не зрозуміло, яке з чисел A або B більше, перед запуском циклу потрібно за необхідності (якщо $A>B$) поміняти їх місцями.

```
if A>B
then begin
  temp:=A; A:=B; B:=temp;
end;
for i:=A to B do
   if isPrime(i) then write(fout,i);
```

Також можна знаходити прості числа з додатковими властивостями. Наприклад, прості числа, у яких всі цифри непарні (11, 19, 113, але не 2, 29, 103), або з сумою цифр, що дорівнює деякому заданому числу k. Для цього необхідно тільки написати ще додаткові функції.

Функція, що перевіряє всі цифри числа на непарність, може бути такою:

```
function oddity(N:longint):boolean;
begin
  while (N mod 2 <> 0) do N:=N div 10;
  if N=0 then oddity:=true
  else oddity:=false;
end;
```

Міркування, за якими вона написана, такі: поки остання цифра числа непарна (число при діленні націло на 2 дає ненульовий залишок), відкидаємо цю цифру. Якщо число врешті-решт перетворилося на 0, усі його цифри були непарні й функція повертає значення **true**, інакше – функція повертає значення **false**.

Друга функція пишеться за таким алгоритмом: поки число не перетворилося на 0, накопичуємо суму цифр (до суми додаємо останню цифру числа) та відкидаємо цю цифру.

```
function Sum_num(N:longint):byte;
varSum:byte;
begin
  Sum:=0;
  while (N <> 0) do
  begin
Sum:=Sum + Nmod 10;  // Накопичення суми цифр
N:=Ndiv 10;   // Відкидання останньої цифри числа
end;
Sum_num:=Sum;
end;
```

Покажемо, як виглядає основна програма для задачі пошуку всіх простих шестицифрових чисел, сума цифр яких дорівнює заданому числу.

```
vark:byte;
i:longint;
fin,fout:text;
begin
assign(fin,'input.txt'); reset(fin);
assign(fout,'output.txt'); rewrite(fout);
  read(fin,k);
for i:=100000 to 999999 do
 if isPrime(i)and(Sum_num(i)=k)
then write(fout,i);
close(fin);
close(fout);
end.
```

Найближче просте число також шукається з використанням функції **isPrime**. Зверніть увагу на те, що нам не відомо, чи найближче просте число менше від заданого, чи більше від заданого. А тому пропонуємо такий алгоритм. Візьмемо наступне **next=N+1** та попереднє **prev=N-1** числа. Далі збільшуємо **next** та зменшуємо **prev** на одиницю, доки вони не стануть простими числами. Відповіддю буде те, яке «ближче» до заданого. Зверніть увагу, що при рівності «відстаней» перевага віддається меншому значенню.

Основна програма пошуку виглядає так:

```
Var N,prev,next:longint;
fin,fout:text;
begin
assign(fin,'input.txt'); reset(fin);
assign(fout,'output.txt'); rewrite(fout);
  read(fin,N);
  next:=N+1;
prev:=N-1;
while not isPrime(next) do inc(next);
 while not isPrime(prev)dodec(prev);
  if  N-prev<=next-N
  then writeln(fout,prev)
  else writeln(fout,next);
close(fin);
close(fout);
end.
```

Нарешті, задача пошуку чисел-близнюків. Нагадуємо, що ***близнюками*** називається пара простих чисел, що відрізняється на 2, наприклад, 3 та 5, 5 та 7, 11 та 13 і т. д. Оскільки пошук чисел-близнюків потрібно здійснювати на заданому діапазоні, організовуємо повноперебірний цикл так само, як уже було вказано раніше. Зверніть тільки увагу на те, що праву границю пошуку потрібно зменшити на 2, оскільки може виявитися ситуація, коли одне число пари входить у вказаний діапазон, а інше – ні.

Указаний цикл повинен бути оформлений таким чином:

```
if A>B
then begin
  temp:=A; A:=B; B:=temp;
end;
for i:=A to B-2 do
   if isPrime(i) and isPrime(i+2)
then writeln(fout,i,' ',i+2);
```

Нарешті, задача «Факторизація». Нагадаємо, що факторизацією називається представлення натурального числа у вигляді добутку простих чисел.

Оскільки множники у розкладанні повинні слідувати в порядку неспадання, пропонується такий алгоритм розв'язання задачі. Доки число не перетворилося на одиницю, перебираємо підряд усі

натуральні числа, починаючи з двох, і ділимо на кожне з них, доки ділення виконується без остачі.

Наприклад, візьмемо число 50. Спочатку ділитимемо це число на 2, доки воно ділиться націло:

50 : 2 = 25.

Далі спробуємо 25 поділити на 3 – не ділиться та на 4 – не ділиться. Наступне натуральне число 5, на яке 25 поділиться двічі:

25 : 5 = 5 : 5 = 1.

Після отримання одиниці процес ділення припиняємо. Отже, результат факторизації:

50 = 2 * 5 * 5.

Цей алгоритм є дуже простим, але нераціональним для великих простих чисел, наприклад, 93283501. У цьому випадку буде поступово здійснюватися перебір усіх чисел від 2 до самого числа 93283501, але при цьому жодного ділення не відбудеться, оскільки задане число само по собі просте. Щоб значно скоротити перебір, знову згадаємо, що складене число має принаймні один дільник до кореня з числа і добавимо цю перевірку в заголовку циклу.

Програма, яка реалізує розкладання числа на прості множники, має вигляд:

```
Var N,divider:longint;
fin,fout:text;
begin
assign(fin,'input.txt'); reset(fin);
assign(fout,'output.txt'); rewrite(fout);
read(fin,N);
divider:=2;
while (N<>1)and(divider<=sqrt(N)) do
begin
while N mod divider = 0 do
begin
write(fout,divider);
         N:=N div divider;
if N>1 thenwrite(fout,'*');
end;
inc(divider);
end;
if N>1 thenwrite(fout,N);
writeln(fout);
close(fin);
close(fout);
end.
```

Мовою C++ розв'язання задачі має такий вигляд:

```
#include<bits/stdc++.h>
usingnamespacestd;
intmain()
{
int N;
freopen(«input.txt»,»r»,stdin);
freopen(«output.txt»,»w»,stdout);
cin>> N;
int i=2;
while (N!=1 && i<=sqrt(N))
    {
if (N%i==0)
       {
cout<<i;
           N=N/i;
if (N!=1) cout<<»*»;
       }
else i++;
    }
if (N!=1) cout<<N;
cout<<endl;
return 0;
}
```

А тепер запропонуємо нестандартну задачу.

Задача «Хвостові нулі у числі N!».

Умова задачі. Задано натуральне число N. Напишіть програму, що визначає, якою кількістю нулів завершується десятковий запис числа $N!$ Наприклад, число $10! = 3628800$ завершується двома нулями.

Вхідні дані. У вхідному файлі записано єдине число N $(2 \le N \le 2^{31} - 1)$.

Вихідні дані. У вихідний файл потрібно вивести єдине число – кількість хвостових нулів у числі $N!$

Ідея розв'язання. Перший варіант розв'язання, який пропонують учні, це так званий «лобовий»: обчислюється значення N!, а потім рахується кількість нулів наприкінці отриманого числа. При всій своїй привабливості цей розв'язок в жодному разі не можна вважати правильним.

Для учнів-початківців пропонуємо спочатку реалізувати програму, яка обчислює перші 25 чисел – факторіали відповідних чисел:

```
var i:longint; factorial:int64;
fout:text;
begin
assign(fout,'output.txt');rewrite(fout);
factorial:=1;
for i:=1 to25do
begin
factorial:=factorial*i;
writeln(fout,i,'!=',factorial);
end;
close(fout);
end.
```

Після запуску цієї програми у вихідному файлі з'явиться така послідовність чисел:

1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
21! = –4249290049419214848
22! = –1250660718674968576

23! = 8128291617894825984
24! = –7835185981329244160
25! = 7034535277573963776

Уважно подивившись на цю послідовність, ви побачите, що після 20!, яке дорівнює великому додатному числу, було отримано 21! – від'ємне число. Але це неможливо! Факторіал не може бути від'ємним числом, оскільки це добуток натуральних чисел. Насправді це результат так званого виходу за межі типу. Отже, обчислюючи факторіал, буде отримано правильний результат тільки для $N \le 20$. Вхідні ж дані передбачають набагато більші значення.

Розглянемо правильну математичну модель задачі. Очевидно, що при множенні послідовності натуральних чисел у результаті отримується 0 у кінці числа, якщо множаться числа 5 і 2, а також числа, в розкладанні яких є двійки та п'ятірки. Оскільки чисел із двійками у розкладанні набагато більше, потрібно враховувати тільки числа, що мають у розкладанні п'ятірки. Причому потрібно враховувати всі п'ятірки у розкладанні.

Отже, алгоритм, що розв'язує вказану задачу, повинен фактично рахувати кількість чисел, що кратні 5, 25, 125, 625 і т. д., що не перевищують заданого числа N. Програма, що реалізує описаний алгоритм, має такий вигляд:

```
var N,answer:longint;pow5:int64;
fin,fout:text;
begin
assign(fin,'input.txt');reset(fin);
assign(fout,'output.txt');rewrite(fout);
read(fin,N);
    pow5:=5;
answer:=0;
while (pow5<=N) do
begin
answer:=answer+N div pow5;
        pow5:=pow5*5;
end;
writeln(fout,answer);
close(fin);
close(fout);
end.
```

Перевірку підрахунку кількості чисел у наведеній реалізації зроблено за допомогою обчислення степеня п'ятірки **pow5**, який не повинен перевищувати заданого числа **N (pow5 ≤ N)**. А в тілі циклу кількість чисел, що кратні відповідному степеню п'ятірки, обчислюється очевидною формулою: **answer:=answer+N div pow5.**

Мовою C++ розв'язок має такий вигляд:

```
#include <bits/stdc++.h>
using namespace std;
int main(){
int N;
freopen(«input.txt»,»r»,stdin);
freopen(«output.txt»,»w»,stdout);
cin>> N;
    long long i=5,answer=0;
    while (i<=N)
    {
        answer+=N/i;
        i*=5;
    }
cout<<answer<<endl;
    return 0;
}
```

Конкурс «Бебрас–2017» у світі та Україні

12–14 листопада 2017 року в Україні вже вдесяте проведено Міжнародний конкурс з інформатики та комп'ютерного мислення «Бобер–2017» для учнів 2–11-х класів. Цього року в конкурсі взяли участь понад 2 мільйони 160 тисяч учнів із 44-х країн світу.

За кількістю учасників конкурсу Україна посіла четверте місце у світі:

№ з/п	**Країна**	**К-ть учасників**
1	Франція	598869
2	Німеччина	341241
3	Велика Британія	143134
4	**Україна**	**117463**
5	Білорусь	111554
6	Тайвань	111162
7	Чехія	74518
8	Словаччина	74216
9	Китай	46053
10	США	44891
11	Сербія	42539
12	Литва	41708
13	Італія	41064
14	Туреччина	35164
15	Австрія	31034
16	Словенія	29993
17	Македонія	24820
18	Польща	24168
19	Росія	22549
20	ПАР	22056

21	Угорщина	21411
22	Нідерланди	18447
23	Швейцарія	16395
24	Канада	15730
25	Хорватія	15247
26	Румунія	15148
27	Латвія	13027
28	Пакистан	11170
29	Швеція	9696
30	Південна Корея	7203
31	В'єтнам	6737
32	Фінляндія	6564
33	Ірландія	5717
34	Японія	5509
35	Боснія і Герцеговина	4164
36	Естонія	3919
37	Індонезія	3714
38	Іран	2615
39	Бельгія	2345
40	Ісландія	2045
41	Болгарія	414
42	Монголія	251
43	Португалія	160
44	Йорданія	155

Слід відзначити, що викладання інформатики в Україні починаючи з другого класу, дало змогу досягти вагомих результатів у конкурсі «Бобер».

За кількістю учасників серед учнів 2–3 класів Україна впевнено посідає перше місце:

№ з/п	**Країна**	**К-ть учасників**
1	**Україна**	**33868**
2	Словаччина	17980
3	Чехія	17163
4	Білорусь	14296
5	Італія	11654
6	Франція	8848
7	Словенія	7929
8	США	7529
9	Велика Британія	7509
10	Німеччина	6561
11	Росія	6112
12	Сербія	5145
13	Польща	3741
14	Пакистан	2788
15	Литва	2542
16	Нідерланди	2490
17	Македонія	2403
18	ПАР	2256
19	Швейцарія	2075
20	В'єтнам	1864
21	Швеція	1569
22	Китай	1278
23	Ірландія	1186
24	Австрія	1179

25	Угорщина	879
26	Хорватія	789
27	Фінляндія	732
28	Ісландія	386
29	Іран	324
30	Південна Корея	184
31	Боснія і Герцеговина	63
32	Йорданія	24

Ще за одним показником Україна значно випереджає інші країни світу:

Кількість шкіл, які взяли участь у конкурсі:

№ з/п	**Країна**	**К-ть шкіл**
1	**Україна**	**4741**
2	Франція	3342
3	Польща	2778
4	Білорусь	2143
5	Німеччина	1898
6	Росія	1095
7	Словаччина	992
8	Велика Британія	664
9	Чехія	657
10	Туреччина	635
11	Литва	567
12	США	499
13	Румунія	457
14	Хорватія	430
15	Сербія	420

16	Канада	412
17	Нідерланди	339
18	Пакистан	321
19	Швейцарія	253
20	Китай	224
21	Македонія	223
22	Швеція	175
23	Угорщина	164
24	Ірландія	116
25	Фінляндія	109
26	Болгарія	96
27	Боснія і Герцеговина	84
28	Естонія	72
29	Бельгія	40
30	Японія	36
31	Ісландія	23
32	Монголія	9

У банк задач, рекомендованих для конкурсу Міжнародним оргкомітетом, були включені задачі таких українських авторів:

Олександра Кібко та Євгенія Савіна (студенти із Запоріжжя, учасники конкурсу «Бобер» у 2009–2014 роках), Сергій Жуковський (Житомирський ОІППО), Тетяна Затилюк (Андрушівська ЗОШ Житомирської області), Марина Криштопа (Бугаївська ЗОШ Полтавської області), Галина Климович (Зарічненський РМК Рівненської області), Лілія Костів і Ростислав Шпакович (Львівський фізико-математичний ліцей).

Крім того, для українського конкурсу були підготовлені задачі такими авторами:

Світлана Васильченко (Запорізька єврейська гімназія «ОРТ-Алєф»), Людмила Старченко (Харківська академія неперервної

освіти), Андрій Мірошниченко (Дніпровська академія неперервної освіти).

В Україні конкурс проходив у **4741** координаційному центрі, школі чи об'єднанні шкіл. У ньому взяло участь **117 463** учні.

Загальна кількість учасників за віковими групами:

Клас	2-й	3-й	4-й	5-й	6-й
Кількість	14917	18951	17827	13372	11907
Клас	7-й	8-й	9-й	10-й	11-й
Кількість	9989	10559	10239	5055	4647

Кількість учасників по регіонах України:

Область	**Кількість**
Вінницька	2466
Волинська	2910
Дніпропетровська	11902
Донецька	5811
Житомирська	2707
Закарпатська	2497
Запорізька	9685
Івано-Франківська	2244
м. Київ	2738
Київська	3191
Кіровоградська	4623
Луганська	1380
Львівська	12576
Миколаївська	2760
Одеська	5988
Полтавська	5974
Рівненська	3150

Сумська	6095
Тернопільська	2803
Харківська	9277
Херсонська	5465
Хмельницька	2583
Черкаська	2924
Чернівецька	3397
Чернігівська	2351

Висловлюємо вдячність обласним координаторам за організацію та забезпечення успішного проведення конкурсу в своїх регіонах:

Слушний Олег Миколайович	Вінницька область
Семенюк Ірина Василівна	Волинська область
Мірошниченко Андрій Анатолійович	Дніпропетровська область
Пилипчук Олена Анатоліївна	Донецька область
Жуковський Сергій Станіславович	Житомирська область
Шаркадій Інна Володимирівна	Закарпатська область
Васильченко Світлана Володимирівна	Запорізька область
Микицей Сергій Михайлович	Івано-Франківська область
Мірошниченко Наталія Михайлівна	м. Київ
Федорчук Валерій Анатолійович	Київська область
Чала Марина Станіславівна	Кіровоградська область
Лобода Володимир Вікторович	Луганська область
Зелез Мирон Михайлович	Львівська область
Гапиченко Галина Євгенівна	Миколаївська область
Мітельман Ігор Михайлович	Одеська область
Шостя Світлана Петрівна	Полтавська область

Буняк Володимир Олександрович	Рівненська область
Павленко Ірина Миколаївна	Сумська область
Кривокульський Любомир Євстахович	Тернопільська область
Старченко Людмила Миколаївна	Харківська область
Сисоєнко Наталя Анатоліївна	Херсонська область
Дрижал Олександр Михайлович	Хмельницька область
Шемшур Вадим Михайлович	Черкаська область
Скрипська Ганна Володимирівна	Чернівецька область
Євтушенко Наталія Василівна	Чернігівська область

Наступний конкурс відбудеться 11–13 листопада 2018 року (інформаційний лист Інституту модернізації змісту освіти МОН України № 22.1/10-1080 від 13.04.2018).

Детальніше про конкурс на сайті **http://bober.net.ua.**

Запрошуємо вчителів та учнів приєднуватись до цікавого масового міжнародного заходу з інформатики.

З М І С Т

Міжнародний конкурс з інформатики та комп'ютерної вправності «Бебрас–2017»

Матеріали школи програмування переможців конкурсу (Львів–2017)

Редактор *А. Б. Грозний*
Комп'ютерне верстання *У. М. Зарицька*

Формат 60x84/16. Ум. друк. арк. 5,60.
Тираж 6200 пр. Зам. № 852.

Видавництво «Аксіома».
вул. Симона Петлюри, 30а, м. Кам'янець-Подільський, 32300.
Тел./факс: (03849) 3 90 06, (067) 381 29 43.
E-mail: aksiomaprint@ukr.net, sales@aksioma.org.ua
Свідоцтво суб'єкта видавничої справи ДК № 1808 від 26.05.2004 р.