

## Вход и изход за задачи от състезателно програмиране

Задачите, дадени на състезания по програмиране обикновено имат вход, състоящ се от няколко тестови примера, за всеки от които програмата трябва да даде отговор. Обикновено входът се чете от стандартния вход и бива няколко вида:

### Вход със зададен брой тестове

**Пресечни точки** (*Вътрешно състезание на НБУ, 11.12.2010*)

Дадени са  $n$  отсечки в равнината, всяка от които е успоредна на координатна ос. Разглеждаме всички двойки отсечки, които може да образуваме от дадените, така че първата отсечка в двойката да е вертикална, а втората – хоризонтална. Напишете програма, която извежда колко са различните пресечни точки на отсечки в така образуваните двойки.

За всеки тестов пример програмата въвежда броя на отсечките  $n$ , следван от данните за отсечките ( $1 < n < 1000$ ). За всяка отсечка се въвеждат по 2 двойки цели числа (в диапазона от 0 до 1000), които задават двойките координати на краищата на отсечката.

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

Следва решението, което нашият отбор предаде:

```
#include <stdio.h>
#include <memory.h>

#define MAX 1005

int A[MAX][MAX];
int t, n, ax, ay, bx, by, color, tmp, intersectionCount;

void init()
{
    //зануляване
    memset(A, 0, sizeof(A));
    intersectionCount = 0;
}

void solve()
{
    //четем броя на отсечките
```

```
scanf("%d", &n);
for(int i = 0;i < n;i++)
{
    //четем координатите на i-тата отсечка
    scanf("%d%d%d%d", &ax, &ay, &bx, &by);
    color = ax == bx ? 1 : 2;
    if (ax > bx) tmp = ax, ax = bx, bx = tmp;
    if (ay > by) tmp = ay, ay = by, by = tmp;
    if (color == 2)
    {
        for(int a = ax;a <= bx; a++)
        {
            if (A[a][ay] == 1)
            {
                intersectionCount++;
                A[a][ay] = 3;
            }
            else if (A[a][ay] != 3)
                A[a][ay] = color;
        }
    }
    else
    {
        for(int a = ay;a <= by; a++)
        {
            if (A[ax][a] == 2)
            {
                intersectionCount++;
                A[ax][a] = 3;
            }
            else if (A[ax][a] != 3)
                A[ax][a] = color;
        }
    }
}

void print()
{
    //извеждаме резултата
    printf("%d\n", intersectionCount);
}

int main()
{
    //четем броя на тестовете
    scanf("%d", &t);

    //за всеки тест
    while(t--)
    {
        init();
        solve();
        print();
    }
}
```

```
}

```

Първото нещо, което прави програмата е да прочете броя на тестовете. За всеки тестов пример изчитаме броя на отсечките, след което координатите им. Идеята на решението е да запълваме клетките на една матрица **A** с 1 за всяка точка (с целочислени координати) за вертикалните отсечки и с 2 за хоризонталните, или обратно. Докато запълваме клетките за текущата отсечка, ако попаднем на точка с различен номер от нейния, то това е пресечна точка и я отбелязваме с 3. Този подход е известен като оцветяване и може да бъде полезен и за други задачи. Много важно е всички глобални променливи и структури за данни (масиви, хеш-таблици и т.н.), които използваме за решение на текущия тест да бъдат инициализирани наново. Ако матрицата **A** не бъде занулявана за всеки тест, тя ще съдържа несъществуващи пресечни точки (от предишните тестове), което води до грешен отговор за текущия тест. Същото се отнася и за променливата `intersectionCount`, в която пазим броя на пресечните точки.

Добре е да се спазва схема на писане подобна на тази в горанта на програмата, а именно:

```
int main()
{
    scanf("%d", &t);
    while(t--) //за всеки тест
    {
        init(); //инициализиране
        read(); //четене на входа
        solve(); //решаване на поредния тест
        print(); //извеждане на резултата
    }
}
```

Тъй като в предложеното решение на задачата с отсечките не е нужно да пазим информация за координатите на всяка отсечка, функциите за четене и решаване на задачата са заменени с една, която **решава задачата, докато чете входа, без да пази изчетения вход**. Този подход е много по-добър и от към памет и от към време за изпълнение от алтернативата да изчетем целия вход, да запазим данните в някакви структури, след което да ги обхождаме. Винаги когато входните данни не са нужни за по-нататъшни изчисления не трябва да бъдат пазени в паметта. Има задачи в които пазенето им в паметта е просто невъзможно, поради огромния размер на входа. Например ако са ни дадени  $10^{10}$  числа, всяко от които е в интервала от  $-10^{18}$  до  $10^{18}$  и ни питат кое е най малкото и най-голямото от тях то очевидно е че не можем да използваме толкова много памет, а и не е нужно. Ето примерно решение, което използва само две променливи за решение на задачата:

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

int n;
long long k, MIN = LLONG_MAX, MAX = LLONG_MIN;

int main()
{
    cin>>n;
```

```

while(n--)
{
    cin>>k;
    if(k > MAX) MAX = k;
    if(k < MIN) MIN = k;
}

cout<<MIN<<" "<<MAX<<endl;
return 0;
}

```

На пръв поглед това решение изглежда много добро, тъй като след прочитането на всяко число единственото, което правим е две сравнения и евентуално присвояване. Входните данни обаче са прекалено много и на тест с размер на входа приблизително 2 GB програмата работи за 338906 милисекунди или почти 6 минути, което е изключително бавно. Тъй като алгоритъма е изключително прост и няма видимо поле за оптимизация, остава да подобрим самото четене. Нека опитаме да изчетем данните като използваме `scanf` вместо `cin`:

```

#include "stdio.h"

int n;
long long k, MIN = LLONG_MAX, MAX = LLONG_MIN;

int main()
{
    scanf("%lld\n", &n);
    while(n--)
    {
        scanf("%lld\n", &k);
        if(k > MAX) MAX = k;
        if(k < MIN) MIN = k;
    }

    printf("%lld %lld\n", MIN, MAX);
    return 0;
}

```

Времето необходимо за изпълнение на програмата е 81920 милисекунди или малко повече от минута. Това е 4 пъти по-добро решение по време за изпълнение!

Винаги използвайте `scanf` и `printf`, вместо `cin` и `cout`!

Ако ограниченията на задачата обаче са например в интервала от  $-10^{50}$  до  $10^{50}$ , то тогава не можем да използваме стандартните типове в C. Има две възможности – сами да си напишем функции за работа с дълги числа или да използваме Java и по-специално класа **BigInteger**. Решението би изглеждал по подобен начин:

```

import java.math.BigInteger;
import java.util.Scanner;

public class MinMax

```

```
{
    private static BigInteger k, max, min;

    public static void main(String[] args)
    {
        min = BigInteger.TEN.pow(50); // 10^50
        max = min.negate(); // -10^50

        Scanner s = new Scanner(System.in);
        // Четем броя на числата
        int n = s.nextInt();

        while (n-- != 0)
        {
            k = s.nextBigInteger();
            if(k.compareTo(max) > 0) max = k;
            if(k.compareTo(min) < 0) min = k;
        }

        System.out.println(String.format("%d %d\n", min, max));
    }
}
```

Това решение използва един много полезен клас за четене на данните – **Scanner**. Този клас предоставя изключително полезни методи и си заслужава да бъде разучен. Другата особеност е използването на класа **BigInteger** за работа с големи цели числа. И този клас също разполага с много удобни методи и може да се окаже много полезен, когато на състезание се даде задача, в която трябва да използваме дълги числа. Ако е нужно да се използват реални числа, то тогава може да използваме класа **BigDecimal**. Препоръчително е използването на класа, вместо сами да си пишете операциите за работа с дълги числа, тъй като писането им ще ви отнеме ценно време, а и е възможно да се допусне бърз по време на писането.

## Вход с маркер за край

Друг възможен вид вход е всеки тест да завършва с някакъв маркер за край. Този маркер може да бъде едно или повече предварително обявени числа. Един подобен вход е: На първия ред на стандартния вход е броя на тестовете. За всеки тест четем три числа. Всеки тест завършва с -1.

```
2
2 3 4
5 8 3
9 4 1
-1
2 6 5
-1
```

Примерно изчитане:

```
#include <stdio.h>
int t, a, b, c;
void solve() {}
void print() {}
void init() {}

void read()
{
    while(true)
    {
        scanf("%d", &a);
        if(a == -1)
            break;

        scanf("%d%d", &b, &c);
        printf("%d %d %d\n", a, b, c);
    }
}

int main()
{
    scanf("%d", &t);
    while(t--)
    {
        init();
        read();
        solve();
        print();
    }
    return 0;
}
```

Прочитаме броя на тестовете и за всеки от тях в безкраен цикъл четем първото число и проверяваме дали то не е маркера за край на теста и ако е така прекратяваме цикъла. Ако не е изчитаме и останалите числа.

## Вход до края на стандартния вход

Да разгледаме отново задачата за търсене на минималното и максималното число, само че този път не ни е зададено колко е броят на числата, а само самите числа. Ето как би изглеждало изчитането на подобен вход:

```
while(!feof(stdin))
{
    scanf("%lld\n", &k);
    if(k > MAX) MAX = k;
    if(k < MIN) MIN = k;
}
```

## Вход до края на реда

Този тип вход е може би най-труден за изчитане от всички до сега. Да разгледаме задача I. АНТИКРИЗИСНИ МЕРКИ, давана на XXII PCOP. Перефразирано условите звучи така: Всеки тестов пример за почва с едно число  $k$ , до края на реда следват неопределен брой числа  $n$ , като  $n < 10001$  и  $k < n + 1$ , всяко в интервала  $a [-10000, 10000]$ . Искане се да се сумират най-големите  $k$  числа. Пример:

Вход	Изход
2 10 15 20 10 25	45
3 2 9 4	15

Най-трудното в тази задача е прочитането на входните данни. Тук трябва правилно да изчетем всички тестове до края на входа и за всеки тест трябва да четем до края на реда.

Следва авторското решение, четящо входа с `getline` и `istringstream`:

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

int k, n, a[100000];
string s;

int compare (const void * a, const void * b)
{
    return (-(int*)a + *(int*)b );
}

void string_to_a()
{
    istringstream instr(s);
    instr >> k;
    int i;
    while(instr >> i)
        a[n++] = i;
}

int main()
{
    while (getline(cin, s))
    {
        n = 0;
        string_to_a();
        qsort (a, n, sizeof(int), compare);
        int sum = 0;
        for (int i = 0; i < k; i++)
            sum += a[i];
        cout << sum << endl;
    }
    return 0;
}
```

Алтернативно решение, четящо входа с `gets` и `strtok`:

```
#include <stdio.h>
#include <string>
using namespace std;

#define MAXN 100001
#define MAXL 10001

int k, n, a[MAXN];
char str[(MAXN * MAXL) + MAXL];

int compare (const void * a, const void * b)
{
    return (*(int*)a + *(int*)b );
}

int main()
{
    while (!feof(stdin))
    {
        gets(str);
        n = 0;
        char *p = strtok(str, " ");
        k = atoi(p);
        p = strtok(NULL, " ");

        while(p != NULL)
        {
            a[n++] = atoi(p);
            p = strtok(NULL, " ");
        }

        qsort (a, n, sizeof(int), compare);
        int sum = 0;
        for (int i=0; i < k; i++)
            sum += a[i];
        printf("%d\n", sum);
    }

    return 0;
}
```

Двете решения са еквивалентни, единствената разлика е в четенето на входа. Второто решение е 4 пъти по-бързо от първото.

Решението на Java, използващо класа `Scanner`:

```
import java.io.*;
import java.util.*;

public class E
{
```



```
public static List<Integer> l;  
public static String line;  
public static Scanner s;  
public static int k, n;  
  
public static void main(String[] args) throws IOException  
{  
    l = new ArrayList<Integer>();  
    s = new Scanner(System.in);  
  
    while (s.hasNextLine())  
    {  
        // Четем един ред  
        line = s.nextLine();  
  
        // Създаваме нов Scanner от прочетения ред  
        Scanner s2 = new Scanner(new StringReader(line));  
  
        if(s2.hasNextInt())  
            k = s2.nextInt();  
  
        l.clear();  
        // Изчитаме числата и ги слагаме в списък  
        while (s2.hasNextInt())  
            l.add(s2.nextInt());  
  
        // Решаваме задачата и отпечатваме решението  
        System.out.println(solve(l));  
    }  
}  
  
public static int solve(List<Integer> nums)  
{  
    Collections.sort(nums, new Comparator<Integer>() {  
        public int compare(Integer o1, Integer o2) {  
            if(o1 > o2) return -1;  
            else if(o1 == o2) return 0;  
            else return 1;  
        }  
    });  
  
    int sum = 0;  
    for (int i = 0; i < k; i++)  
        sum += nums.get(i).intValue();  
    return sum;  
}  
}
```

## Пренасочване на входа и изхода

Пренасочването на входа и изхода е много удобен начин за тестване по време на писането на задачата. Може да се наложи да дебъгвате и след отстраняване на грешките да тествате

програмата си. Препоръчително практика, спестяваща доста време в такива ситуации е да създадете файл с входните тестове и да пренасочвате входа, вместо всеки път да го набирате ръчно. Друго предимство на този подход е че пренасочвайки входа ще видите как вашата програма би се държала, когато проверяващия я тества, тъй като проверката става с пренасочване на входа и изхода.

Достатъчно е в началото на `main` функциите на горните C/C++ програми да се поставят следните реове, за пренасочване стандартния вход/изход да е от файл, а не от клавиатурата:

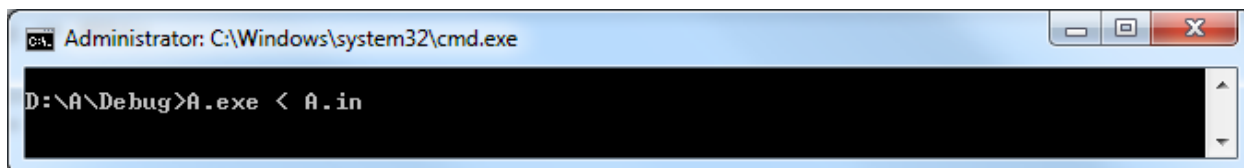
```
freopen("filename.in", "r", stdin);  
freopen("filename.out", "w", stdout);
```

За Java:

```
System.setIn(new FileInputStream("filename.in"));  
System.setOut(new PrintStream(new FileOutputStream("filename.out")));
```

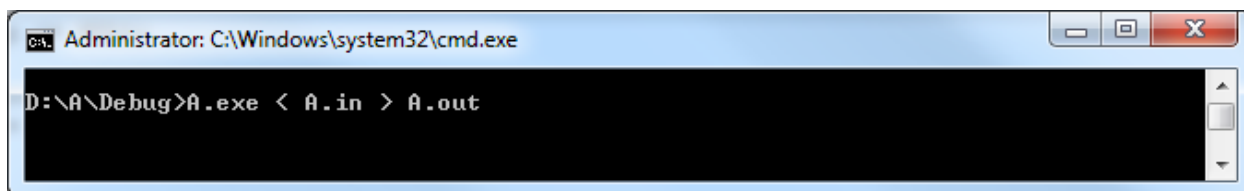
Всеки път преди предаване на решението трябва да помним, че **тези редове трябва да бъдат коментирани**, в противен случай програмата ще чака да чете данни от файл, а не от стандартния вход.

Друг начин за пренасочване без да пишем код (който после трябва да махнем) е да използваме конзолата. Да предположим че в директорията `D:\A\Debug` се намира изпълнимия файл с име `A.exe` и сме подготвили файл с тестове с име `A.in`. Пренасочването на входа за C/C++ програми става по следния начин:



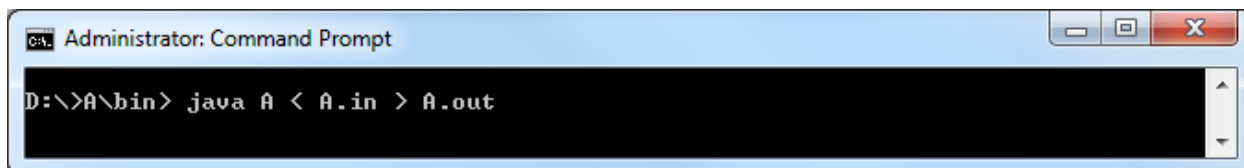
```
Administrator: C:\Windows\system32\cmd.exe  
D:\A\Debug>A.exe < A.in
```

Ако искаме да пренасочим и изхода във файл с име `A.out`:



```
Administrator: C:\Windows\system32\cmd.exe  
D:\A\Debug>A.exe < A.in > A.out
```

Пренасочването на входа и изхода за Java програми става по следния начин:



```
Administrator: Command Prompt  
D:\>A\bin> java A < A.in > A.out
```