Задача 1. Китайская Теорема об Остатках

Источник:	базовая
Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	разумное
Ограничение по памяти:	разумное

Требуется применить теорему в случае взаимно простых модулей.

Формат входных данных

В первой строке задано одно целое положительное число k — количество модулей. Во второй строке дано k целых чисел M_1, M_2, …, M_k — модули. В третьей строке дано k целых чисел A_1, A_2, …, A_k — остатки от деления.

Гарантируется, что:

1. все модули $2 \leqslant M_i \leqslant 10^9$,
2. произведение всех модулей M_i не превышает 10^{18}.
3. все остатки $0 \leqslant A_i < M_i$,
4. любые два модуля M_i и M_j взаимно простые (при $i \neq j$).

Формат выходных данных

Требуется вывести одно целое неотрицательное число X, такое что $X \mod M_i = A_i$ для всех i. Поскольку решений несколько, выведите минимальное неотрицательное число X.

Пример

input.txt
2 100 11 45 9
output.txt
845

input.txt
15 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 1 0 4 2 10 10 11 12 17 10 21 19 40 24 35
output.txt
340973976622192209

Задача 2. Арифметика по модулю

Источник:	базовая II
Имя входного файла:	--
Имя выходного файла:	--
Ограничение по времени:	5 секунд
Ограничение по памяти:	разумное

Код задачи: `2/2_modular`

В этой задаче нужно реализовать арифметику в поле вычетов (т.е. арифметику по простому модулю). Она должна быть объявлена в файле `modular.h` и реализована в файле `modular.c`. Кроме того, в файле `main.c` нужно написать код для тестирования этой арифметики.

Вот простейший пример использования арифметики:

```
#include "modular.h"     //в этом хедере должны быть объявления
#include <assert.h>

int main() {
    MOD = 13;            //устанавливаем глобальный модуль
    int a = 45;
    a = pnorm(a);
    assert(a == 6);
    int x = pmul(padd(7, psub(2, 3)), 5);
    assert(x == 4);
    int y = pdiv(7, x);
    assert(pmul(x, y) == 7);
    MOD = 2;             //меняем модуль на другой
    assert(pnorm(5) == 1);
    return 0;
}
```

Модуль должен храниться в глобальной переменной `MOD`, которая должна иметь тип `int`. В эту переменную записывается модуль в самом начале работы, кроме того, его можно переприсваивать сколько угодно раз в дальнейшем. Модуль должен быть простым числом не более 10^9 — другие значения устанавливать нельзя. Переменная `MOD` должна быть объявлена в `modular.h` и определена в `modular.c`.

В модульной арифметике должно быть 5 функций:

1. Функция `pnorm`: принимает одно значение типа `int`, возвращает `int`. Функция возвращает остаток от деления переданного аргумента по текущему модулю. Входное число по абсолютной величине не превышает 10^9, может быть отрицательным. Выходное значение должно быть в диапазоне от 0 до `MOD - 1`.

2. Функции `padd`, `psub`, `pmul`, `pdiv`: каждая принимает два параметра типа `int`, возвращает `int`. Они реализуют сложение, вычитание, умножение и деление соответственно в поле вычетов по модулю текущего `MOD`. Значения аргументов и выходное значение должны быть в диапазоне от 0 до `MOD - 1`.

Все эти функции должны быть объявлены в `modular.h` и определены в `modular.c`.

Для тестирования написанного кода следует использовать файл `main.c`, где нужно определить точку входа `main`. В функции `main` следует написать какой-то код, чтобы убедиться, что написанный в `modular` код работает правильно. В этом коде для проверки условий используйте встроенную функцию `assert` (в системе тестирования они **не** удаляются). По сути, приведённый выше кусок кода является хорошим примером содержимого `main.c`, только проверок лучше добавить побольше.

При сборке воедино программы из `main.c`, `modular.h` и `modular.c` должен получаться исполняемый файл, который при выполнении запускает ваши тесты, т.е. проверяет правильность кода `modular.c`. Вам нужно отправить в систему тестирования все эти три файла. В системе часть файлов будет заменяться на файлы жюри, в частности:

1. Будет проверяться ваша реализация `modular` с помощью файла `main.c` от «жюри».
2. Будет проверяться, что ваши тесты в `main.c` отлавливают простейшие ошибки в `modular` — то есть файлы `modular` будут подменяться на неправильно работающие.

Гарантируется, что тестирующий код жюри использует ваши функции и переменные корректно, согласно описанным выше условиям и соглашениям. Аналогично, ваш тестирующий код должен также соблюдать все эти условия.

Задача 3. Я милого узнаю по походке...

Источник:	базовая II
Имя входного файла:	`--`
Имя выходного файла:	`stdout`
Ограничение по времени:	разумное
Ограничение по памяти:	разумное

Код задачи: `2/3_whois`

К сожалению, разные компиляторы имеют разные "особенности", и иногда программисту приходится писать отдельный код для некоторых из них. Для этого используется условная компиляция: с помощью `#if` можно сделать, чтобы часть кода компилировалась только на каком-то из компиляторов. Разумеется, для этого нужно уметь узнавать в программе, какой компилятор и с какими настройками сейчас запущен.

В данной задаче предлагается написать простую программу, которая при запуске будет писать, каким компилятором её собрали.

Формат выходных данных

В первой строке должно быть написать имя компилятора. Может быть `MSVC`, `GCC`, `clang` и `TCC`. Во второй строке — битность собранной программы: 32-битная или 64-битная. В третьей строке нужно написать, были ли включены assert-ы при сборке, или они все были удалены из кода. Следует выводить информацию точно в таком же формате, как в примерах.

Примеры

`--`	stdout

	`Compiler: MSVC`
	`Bitness: 64`
	`Asserts: disabled`

	`Compiler: TCC`
	`Bitness: 32`
	`Asserts: enabled`

Комментарий

Рекомендуется поискать в интернете "predefined macros" для разных компиляторов. Обычно для каждого компилятора есть отдельная страница, на которой они указаны. Также могут помочь ответы на stackoverflow.

Кроме того, рекомендуется поставить локально все четыре компилятора для тестирования.

Внимание: Будьте осторожны с компилятором clang! Он немного притворяется другими компиляторами, такими как GCC и MSVC. Подробности: `https://stackoverflow.com/questions/38499462/how-to-tell-clang-to-stop-pretending-to-be-other-compilers`

Задача 4. Максимальный отрезок

Источник:	основная* I
Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	разумное
Ограничение по памяти:	разумное

Код задачи: `2/4_maxseg`

Дан массив A длины N, все элементы массива A_i **неотрицательные**.

Далее нужно ответить на M вопросов. В каждом вопросе задана позиция L в массиве и число S. Нужно найти самый длинный отрезок с левым краем в L, сумма элементов которого **не** превышает S. Вывести требуется правый край R такого отрезка.

В данной задаче края отрезка задаются так, как принято в языке C, то есть:

1. Элементы массива нумеруются начиная с нуля.
2. Отрезок с краями $L < R$ включает L-ый элемент, но не включает R-ый элемент. Если $L = R$, то отрезок пустой.

В первой строке дано два целых числа: N — количество элементов массива A и M — количество вопросов ($1 \leqslant N, M \leqslant 10^5$).

В следующих N строках записаны элементы массива A_i, по одному в строке. Все эти числа целые неотрицательные и не превышают 10^9.

В оставшихся M строках записаны вопросы, по одному в строке. Вопрос описывается двумя целыми числами: L — левый край искомого отрезка и S — насколько большой может быть сумма элементов отрезка ($0 \leqslant L \leqslant N$, $0 \leqslant S \leqslant 10^{15}$).

Использование хедеров в этой задаче запрещается. Решение задачи должно быть разбито на три исходных файла: `sum.c`, `query.c` и `main.c` Ровно эти файлы следует отправлять в систему тестирования.

В файле `sum.c` должны быть определены следующие функции:

```
//выполнить подготовку массива: запускается один раз перед запусками Sum
//здесь arr[0..n) -- это массив A из входного файла
void Init(const int *arr, int n);
//найти сумму элементов A на отрезке [l..r)
//использует данные, посчитанные функцией Init
int64_t Sum(int l, int r);
```

Используя эти функции, нужно реализовать в файле `query.c` ещё одну функцию:

```
//находит самый длинный отрезок с началом в l и суммой не более sum
//возвращает правый край искомого отрезка
int Query(int l, int64_t sum);
```

В файле `main.c` должна быть функция `main` (точка входа), чтение и запись данных, вызовы вышеописанных функций.

Можно использовать глобальные и статические переменные для хранения любых данных.

Пример

input.txt	output.txt
10 7 1 4 0 5 6 0 0 1 5 3 0 100 0 5 4 11 4 12 4 13 10 100 8 0	10 3 8 9 9 10 8

Комментарий

Можно реализовать функцию `Init` за время $O(N)$, так что функция `Sum` будет работать за время $O(1)$. Для этого надо предподсчитать суммы на $[0, k)$ для всех k, а сумму на отрезке вычислять через две такие суммы.

Заметим, что сумма элементов массива на отрезке $[l, r)$ аналогична интегралу функции на отрезке $[a, b]$. В таком случае идея данной задачи является дискретным аналогом формулы Ньютона-Лейбница =)

Задача 5. Разложение на простые

Источник:	основная I
Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	разумное
Ограничение по памяти:	разумное

Код задачи: `2/5_primefact`

Дано несколько натуральных чисел, нужно разложить каждое число X на простые множители. Искомое разложение имеет вид:

$$X = p_1^{t_1} * p_2^{t_2} * \ldots * p_k^{t_k}$$

При этом все числа p_i должны быть простыми и упорядоченными строго по возрастанию, а числа t_i должны быть целыми положительными.

Требуется реализовать следующую функцию для поиска разложения:

```
//структура, в которой хранится разложение на простые множители
typedef struct Factors {
    int k;              //сколько различных простых в разложении
    int primes[32];     //различные простые в порядке возрастания
    int powers[32];     //в какие степени надо эти простые возводить
} Factors;

//функция, которая находит разложение числа X и записывает его в структуру res
void Factorize(int X, Factors *res);
```

Использование хедеров в этой задаче запрещается. Функция `Factorize` должна быть реализована в исходном файле `factorize.c`. Всё остальное (точка входа, чтение/запись и пр.) должно быть в исходном файле `main.c`. Отправлять нужно только эти два исходных файла.

Учтите, что файлы в вашей программе будут частично подменяться на файлы “жюри”, чтобы проверить, что вы действительно разделили программу на части как описано.

Формат входных данных

В первой строке дано целое число M — количество запросов ($1 \leqslant M \leqslant 1\,000$). В каждой из остальных M строк записано одно целое число X ($1 \leqslant X \leqslant 10^9$).

Формат выходных данных

Для каждого числа X нужно вывести его разложение на простые множители в отдельной строке. Формат вывода должен быть полностью аналогичен формату примера. Сначала пишется число X, потом знак равенства. Если простых в разложении нет ($k = 0$), то надо написать просто единицу. Иначе нужно вывести все степени в виде `p^t`, разделённые знаком умножения. Между соседними токенами должен стоять ровно один пробел.

Пример

input.txt	output.txt
15 1 2 3 4 5 6 7 8 9 10 1000000000 999999999 999999987 999999997 999999937	1 = 1 2 = 2^1 3 = 3^1 4 = 2^2 5 = 5^1 6 = 2^1 * 3^1 7 = 7^1 8 = 2^3 9 = 3^2 10 = 2^1 * 5^1 1000000000 = 2^9 * 5^9 999999999 = 3^4 * 37^1 * 333667^1 999999987 = 3^1 * 7^2 * 6802721^1 999999997 = 71^1 * 2251^1 * 6257^1 999999937 = 999999937^1

Задача 6. Численное дифференцирование

Источник:	основная I
Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	разумное
Ограничение по памяти:	разумное

Код задачи: `2/6_numdiff`

Дана гладкая функция $f(x)$, определённая на отрезке $[0, 1]$, и набор точек x_i. Требуется вычислить производную $\frac{df}{dx}(x)$ в заданных точках x_i.

На каждом тесте ваша программа будет собираться заново. К списку отправленных вами исходных файлов будет добавляться файл `func.c`, в котором определена одна программная функция `Function`. Она вычисляет значение $f(x)$ по заданному аргументу x.

Например, так выглядит содержимое func.c на первом тесте:

```
#include <stdlib.h>

double Function(double x) {
    if (x < 0.0 || x > 1.0)
        exit(666);       //out of range
    return x * x - x;
}
```

Остальные входные данные (точки x_i) задаются в файле `input.txt`.

Формат входных данных

В первой строке дано целое число M — количество точек ($1 \leqslant M \leqslant 1\,000$). В каждой из остальных M строк записано одно вещественное число x_i ($0 \leqslant x_i \leqslant 1$).

Формат выходных данных

Для каждого числа x_i нужно вывести значение производной $f'(x_i)$. Значение считается верным, если его абсолютная или относительная погрешность не превышает $3 \cdot 10^{-6}$. Тест засчитывается, если все вычисленные значения верные.

Рекомендуется выводить все значения с максимальной точностью.

Пример

input.txt	output.txt
6 0.0 0.2 0.4 0.6 0.8 1.0	-1.000000000000000 -0.600000000000000 -0.200000000000000 0.200000000000000 0.600000000000000 1.000000000000000

Комментарий

Для решения задачи вспомните определение производной.

Задача 7. Количество простых

Источник:	основная II
Имя входного файла:	--
Имя выходного файла:	--
Ограничение по времени:	полторы секунды*
Ограничение по памяти:	разумное

Код задачи: `2/7_primes`

В этой задаче нужно опять написать код для поиска простых чисел. Требуется отправить на проверку два файла: `primes.c` и `main.c`. Файл `primes.c` должен включать (`#include`) в себя файл `primes.h` и реализовывать объявленные в нём три функции. Файл `main.c` должен содержать функцию `main`, в которой тестируются объявленные в `primes.h` функции с использованием `assert`. Сам файл `primes.h` отправлять не обязательно, он будет автоматически добавлен при проверке (изменять его содержимое вы **не** можете).

Содержимое хедера `primes.h` таково:

```
#ifndef PRIMES_9183746069462
#define PRIMES_9183746069462

//returns: 1 if x is prime number, 0 otherwise
int isPrime(int x);
//returns minimal prime number p such that p >= x
int findNextPrime(int x);
//returns the number of primes x such that l <= x < r
int getPrimesCount(int l, int r);

#endif
```

Гарантируется, что числа `x`, `l`, `r` лежат в диапазоне от 0 до 10^7 включительно. Кроме того, при вызове `getPrimesCount` выполняется $l \leqslant r$. Гарантируется, что за запуск программы тестирующий код делает не более $2 \cdot 10^6$ вызовов функций. Эти условия соблюдает тестирующий код жюри, и ваш тестирующий код тоже должен их соблюдать.

При проверке будет добавлен хедер `primes.h`. Кроме того, `main.c` и `primes.c` будут поочерёдно подменяться на файлы жюри, чтобы проверить правильность работы функций и качество тестирования.

Внимание: Единица трансляции `primes.c` **не** должна определять никаких публичных символов, кроме описанных трёх функций. Если вам нужно завести ещё какую-то функцию или глобальную переменную, сделайте её приватной для единицы трансляции.

Комментарий

Следует использовать решето Эратосфена для поиска простых чисел.

Известно, что среди чисел порядка N примерно каждое $(\ln N)$-ое является простым — об этом гласит Prime Number Theorem. Кроме того, расстояние между соседними простыми называется prime gap и растёт примерно логарифмически.

Задача 8. Переменное количество аргументов

Источник:	основная II
Имя входного файла:	--
Имя выходного файла:	--
Ограничение по времени:	3 секунды
Ограничение по памяти:	разумное

Код задачи: `2/8_vararg`

Цель данной задачи — научиться использовать и писать функции с переменным количеством аргументов. От вас требуется реализовать функции, объявленные в хедерах `logger.h` и `pack.h`, в исходных файлах `logger.c` и `pack.c` соответственно. Эти два файла и надо отправить на проверку.

Рекомендуется скачать и посмотреть архив с файлами `logger.h`, `pack.h` и `sample.c`.

Хедер `logger.h` объявляет три функции для записи сообщений в лог:

```
void logSetFile(FILE *file);
void logPrintf(const char *format, ...);
int getLogCallsCount();
```

Подробное описание этих функций содержится в полной версии хедера, которую можно скачать выше по ссылке. Текущий лог-файл и текущее количество успешных вызовов можно хранить в глобальных переменных. В функции `logPrintf` рекомендуется просто перенаправить вызов в соответствующую функцию из `stdio.h`.

Хедер `pack.h` объявляет функцию записи (сериализации) простых данных в байтовый буфер:

```
int pack(char *buffer, const char *format, ...);
```

Подробное описание этой функции содержится в полной версии хедера, которую можно скачать выше по ссылке.

При проверке будут добавлены хедеры и тестирующий код жюри.

Если есть вопросы по требуемому поведению функций в каких-либо случаях, задавайте вопросы.

Пример использования функций можно видеть в этом коде (файл `sample.c`):

```c
#include "logger.h"
#include "pack.h"
#include <stdlib.h>
#include <string.h>
#include <assert.h>

int main() {
    logPrintf("Not enabled logging yet\n"); //ignored
    logSetFile(stderr);
    logPrintf("Logging in stderr\n");       //goes to stderr
    logSetFile(stdout);
    logPrintf("Logging enabled!\n");        //goes to stdout

    int five = 5, ten = 10;
    double unit = 1.0;
    const char *hello = "hello";

    logPrintf(
        "Quering number of bytes for: [%d] [%lf] [%s] [%d]\n",
        five, unit, hello, ten
    );
    int bytes = pack(NULL, "%d%lf%s%d", five, unit, hello, ten);
    logPrintf("Allocating buffer of size %d\n", bytes);
    char *buffer = malloc(bytes);
    logPrintf("Packing data into buffer\n");
    int written = pack(buffer, "%d%lf%s%d", five, unit, hello, ten);

    logPrintf("Checking result\n");
    assert(written == bytes && written == 22);
    char correct[22] = {                    //note: assume little-endian
        0x05, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xF0, 0x3F,
        'h', 'e', 'l', 'l', 'o', 0,
        0x0A, 0x00, 0x00, 0x00
    };
    assert(memcmp(buffer, correct, sizeof(correct)) == 0);

    free(buffer);
    logSetFile(0);
    logPrintf("Sample finished\n");         //ignored

    logSetFile(stdout);
    logPrintf("Sample really finished\n");  //printed

    assert(getLogCallsCount() == 7);
    return 0;
}
```

Задача 9. Арифметическая прогрессия

Источник:	повышенной сложности II
Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	разумное
Ограничение по памяти:	разумное

Код задачи: `2/9_arithm`

Дан массив из N беззнаковых 32-битных целых чисел. Требуется найти в нём максимальный по длине отрезок, который является арифметической прогрессией.

В данной задаче имеется ввиду арифметическая прогрессия по модулю 2^{32}. В такой арифметической прогрессии каждое следующее число получается из предыдущего прибавлением фиксированного X по модулю 2^{32}. Следует напомнить, что в языке C операции сложения, вычитания и умножения над беззнаковыми числами автоматически выполняются по этому модулю.

Например, следующая последовательность является арифметической: 2294967296, 3294967296, 0, 1000000000.

В вашей программе два момента должны настраиваться при помощи передаваемого компилятору макроса:

- По умолчанию файловый ввод/вывод должен быть текстовым. Если же при компиляции определён макрос `BINARY_IO`, то ввод/вывод должен быть бинарным.
- По умолчанию в вашем коде должна быть определена точка входа `main`. Если же при компиляции установлен макрос `NO_MAIN`, то функция `main` должна отсутствовать, а выполнение всей работы должно запускаться вызовом функции `DoAll` (без параметров и возвращаемого значения). В этом случае точка входа будет определена во внешней единице трансляции.

Формат входных данных

Если режим текстовый, то в файле задано N целых чисел, разделённых пробелами и/или переводами строк. Если режим бинарный, то в файле задано N 32-битных целых чисел с порядком байтов little-endian.

Количество чисел N может изменяться от 2 до 10^6. Каждое число лежит в диапазоне от 0 до $2^{32} - 1$ включительно.

Формат выходных данных

Нужно вывести следующие целые числа (в этом порядке):

- K — количество элементов в найденной арифметической прогрессии,
- L — номер первого элемента, который входит в прогрессию,
- R — номер первого элемента после прогрессии (т.е. который **уже** не входит),
- сама найденная арифметическая прогрессия (K чисел).

Элементы нумеруются, начиная с нуля.

Если режим текстовый, то нужно выводить целые числа по одному в строке. Если режим бинарный, то нужно выводить числа как 32-битные целые с little-endian порядком байт.

Если оптимальных решений несколько, требуется выбрать среди них решение с минимальным L.

Пример

input.txt	output.txt
4 1 3 5 7 4 1 3 5 6 7 8 7 5	4 1 5 1 3 5 7
5 3 1 0 4294967295 4294967294 4294967293 4294967292	6 2 8 1 0 4294967295 4294967294 4294967293 4294967292

Комментарий

В первом примере найдена прогрессия 1 3 5 7, которая начинается с элемента номер 1, и идёт вплоть до элемента номер 5 (исключая этот элемент).

Примеры в бинарном формате можно скачать здесь.

Задача 10. Безумная линковка

Источник:	повышенной сложности I
Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	разумное
Ограничение по памяти:	разумное

Код задачи: `2/A_crazylink`

Эта задача стремится показать, что линкер — это слепое чудовище! В закрытой для вас единице трансляции есть N символов с известными именами, но с неизвестными типами. Тип этих символов сообщается только во время запуска программы (задан во входном файле). Вам нужно научиться оперировать с этими символами в таких условиях.

На каждом тесте ваша программа будет собираться заново. К списку отправленных вами исходных файлов будет добавляться файл `symbols.c`, в котором определено ровно 10 символов с именами от `symbol0` до `symbol9`. Далее эта программа должна скомпилироваться и слинковаться в исполняемый файл. Затем этот исполняемый файл запускается, читает из входного файла типы символов и запросы, и выводит ответы в выходной файл.

Формат входных данных

В первой строке входного файла записано целых числа: N — сколько символов используется в тесте и M — количество запросов ($1 \leqslant N \leqslant 10$, $1 \leqslant M \leqslant 100$). Среди определённых в `symbols.c` десяти символов в тесте используются только первые N штук, остальные не упоминаются.

В следующих N строках задаются типы символов, по одному символу в строке. Символы описываются строго в порядке от 0-ого до $(N-1)$-ого. Типы описываются в формате языка C. Если символ является переменной, то записано определение переменной, а если функцией — то прототип функции.

Возможны следующие типы:

- переменная типа `int` или `double`;
- переменная-указатель типа `int*` или `double*`;
- функция с количеством аргументов от нуля до двух; в этом случае каждый аргумент имеет тип `int` или `double`;
- переменная типа «указатель на функцию» с ограничениями как в предыдущем пункте;

Для вашего удобства при форматировании типов жёстко соблюдается единый стиль. У параметров функции и указателя на функцию отсутствуют имена. Если функция или указатель на функцию не принимает параметров, то в скобках нет ничего (в частности, не пишется `void`). Все переменные/функции имеют имя, совпадающее с именем соответствующего символа. В обычном указателе звёздочка ставится вплотную к имени переменной. Одиночный пробел ставится после запятой, а также после основной части типа переменной или возвращаемого типа функции. Других пробелов нет.

В последних M строках описываются запросы, по одному в строке. Каждый запрос — это последовательноть целых чисел длиной от одного до трёх. Все числа последовательности лежат в диапазоне от 0 до $N-1$ включительно.

Бывает два типа запросов:

1. Найти значение переменной: в этом случае задаётся номер символа-переменной, значение которого надо вывести.

2. Вычислить значение функции: в этом случае сначала задаётся номер символа-функции, которую надо вызвать, а потом номера символов-переменных, которые надо передать в функцию в качестве аргументов.

Если символ является указателем на переменную или указателем на функцию, то надо предварительно разыменовать эту переменную. Гарантируется, что все запросы сформированы корректно, количество аргументов и их типы подходят (без преобразований).

Формат выходных данных

Нужно вывести M строк, в каждой строке ответ на соответствующий запрос. Ответом считается результат вычисления, который получается или типа `int`, или типа `double`. Значения типа `double` рекомендуется распечатывать с максимальной точностью.

Запросы нужно вычислять в порядке их описания: некоторые функции могут иметь побочные эффекты.

Пример

Содержимое файла `symbols.c` для примера:

```
int rnd(double x) { return (int)x; }
int symbol0 = 42;
double symbol1 = 5.7;
int* symbol2 = &symbol0;
int symbol3() { return -1; }
int symbol4(int a) { return a * 4; }
int symbol5(int a, int b) { return a - b + 7; }
double symbol6(double a) { return a * a - 3.5; }
int (*symbol7)(double a) = &rnd;
int symbol8 = 23;
int symbol9;
```

input.txt	output.txt

9 10
int symbol0;
double symbol1;
int *symbol2;
int symbol3();
int symbol4(int);
int symbol5(int, int);
double symbol6(double);
int (*symbol7)(double);
int symbol8;
0
1
2
3
4 0
8
5 0 8
6 1
7 1
5 0 0
	42
	5.700000000000000
	42
	-1
	168
	23
	26
	28.990000000000002
	5
	7

Комментарий

Данная задача крайне далека от реальности. В реальности если кто-то полагается на несовпадающие типы при линковке, надо сразу сослать его в Сибирь!

В зависимости от способа реализации в решении может получиться очень много кода. Рекомендуется подумать, как минимизировать объём кода и вероятность облажаться. У меня получилось 163 строки.

Учтите, что к этой задаче подключены компиляторы Visual C, GCC, Clang и TCC — решение должно работать на них всех.

Задача 11. Призрак Старого Парка +

Источник:	повышенной сложности
Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	разумное
Ограничение по памяти:	разумное

Нужно решить головоломку из компьютерной игры «Призрак Старого Парка» на полях большего размера.

Формат входных данных

В первой строке задано одно целое число N — размер поля ($1 \leqslant N \leqslant 200$).

В остальном формат входных данных такой же, как в предыдущей задаче про головоломку.

Формат выходных данных

Формат выходных данных такой же, как в предыдущей задаче про головоломку.

Пример

input.txt	output.txt
5 ***** ***** ***** ***** *****	15 3 3 1 2 4 2 3 4 5 3 4 4 1 1 3 5 2 5 5 5 2 4 4 3 2 1 5 2 2 2
4 **.. ..*. **** .*..	4 2 2 4 4 4 3 2 1

Комментарий

Первый пример показывает, как решить оригинальную задачу из игры. Он совпадает с первым тестом. Второй пример **не** совпадает со вторым тестом.

Подсказка: попробуйте сократить количество неизвестных до $O(N)$.

Задача 12. Китайская Теорема об Остатках +

Источник:	повышенной сложности
Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	2 секунды
Ограничение по памяти:	разумное

Требуется применить теорему в случае взаимно простых модулей.

Искомое число может получиться очень **длинным**.

Формат входных данных

В первой строке задано одно целое положительное число k — количество модулей. Во второй строке дано k целых чисел M_1, M_2, …, M_k — модули. В третьей строке дано k целых чисел A_1, A_2, …, A_k — остатки от деления.

Гарантируется, что:

1. все модули $2 \leqslant M_i \leqslant 10^9$,
2. количество модулей $k \leqslant 2\,000$,
3. все остатки $0 \leqslant A_i < M_i$,
4. любые два модуля M_i и M_j взаимно простые (при $i \neq j$).

Формат выходных данных

Требуется вывести одно целое неотрицательное число X, такое что $X \mod M_i = A_i$ для всех i. Поскольку решений несколько, выведите минимальное неотрицательное число X.

Пример

input.txt
2 100 11 45 9
output.txt
845

input.txt
15 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 1 0 4 2 10 10 11 12 17 10 21 19 40 24 35
output.txt
340973976622192209