Задача 1. Простая сортировка

Источник:	базовая*
Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	1 секунда
Ограничение по памяти:	разумное

В первой строке записано целое число N — количество записей ($1 \leqslant N \leqslant 10\,000$). В остальных N строках содержатся записи, по одной в строке.

Для каждой записи указаны ключ и значение через пробел. Ключ — это целое число в диапазоне от 0 до 10^9 включительно, а значение — это строка от одного до семи символов включительно, состоящая только из маленьких букв латинского алфавита.

Требуется вывести ровно те же самые N записей, но в другом порядке. Записи должны быть упорядочены по возрастанию ключа. Если у нескольких записей ключ равный, то нужно упорядочить их в том порядке, в котором они встречаются по входном файле.

Пример

input.txt	output.txt
7 3 qwerty 3 string 6 good 1 a 3 ab 2 hello 5 world	1 a 2 hello 3 qwerty 3 string 3 ab 5 world 6 good

Пояснение к примеру

В примере 7 записей с ключами 1, 2, 3, 5 и 6 — именно в таком порядке записи и выведены в выходном файле. Обратите внимание, что есть три записи с ключом 3: `qwerty`, `string`, `ab`. Они выведены ровно в том порядке, в котором они идут во входном файле.

Задача 2. Битовый массив

Источник:	базовая*
Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	1 секунда*
Ограничение по памяти:	**5 мегабайт**

Память современных компьютеров можно представить в виде очень длинного массива битов, в котором каждый бит может принимать значения 0 или 1. Однако возможность обращаться к конкретному биту по его номеру отсутствует: минимальная адресуемая единица памяти — это байт, обычно состоящий из 8 битов. Чтобы работать с отдельными битами памяти, необходимо обращаться к байтам (или словам), которые их содержат, и применять различные битовые операции.

В данной задаче нужно реализовать структуру данных “битовый массив” / “bitset”. Это массив, элементы которого принимают лишь значения 0 и 1. Эти элементы можно легко сохранить в массиве байтов, храня одно значение в каждом байте. Однако если сохранять по 8 элементов в байте, то снижается потребление памяти и появляется возможность ускорить некоторые операции — именно это и требуется сделать в данной задаче.

Нужно реализовать набор функцию со следующей сигнатурой:

```
//какой-нибудь целочисленный тип (желательно беззнаковый)
typedef ??? bitword;

//инициализирует массив битов длины num, устанавливая все его биты в ноль
void bitsetZero(bitword *arr, int num);
//возвращает значение idx-ого бита (0 или 1)
int bitsetGet(const bitword *arr, int idx);
//устанавливает значение idx-ого бита в newval (которое равно 0 или 1)
void bitsetSet(bitword *arr, int idx, int newval);
//возвращает 1, если среди битов с номерами k
//для left <= k < right есть единичный, и 0 иначе
int bitsetAny(const bitword *arr, int left, int right);
```

Массив слов `arr` управляется вызывающим, и может указывать, к примеру, на глобальный массив достаточного размера. Вы можете самостоятельно выбрать базовый тип для слова `bitword`. Однобайтовый тип подойдёт, однако для лучшей производительности рекомендуется использовать слова большего размера. Функция `bitsetAny` определяет, есть ли в заданном окне/отрезке битов хотя бы один бит, равный единице. При реализации этой функции **необходимо** быстро обрабатывать окна большого размера: для этого нужно целиком перебирать слова, полностью попадающие внутрь окна, не перебирая отдельные биты.

При помощи реализованной структуры нужно решить тестовую задачу.

Формат входных данных

В первой строке записано целое число N — количество операций, которые нужно обработать ($1 \leqslant N \leqslant 10^5$). В каждой из следующих N строк описывается одна операция над битовым массивом.

Описание операции начинается с целого числа t, обозначающего тип операции. Если $t = 0$, то это операция `bitsetZero`, и вторым целых числом в строке указан размер массива `num`. Если $t = 1$, то это операция `bitsetGet`, и в строке также записано целое число `idx` — номер бита. Значение этого бита нужно выдать в выходной файл. Если $t = 2$, то это операция

`bitsetSet`, и в строке также содержатся целые числа `idx` и `newval`. Здесь `idx` — номер бита, который нужно изменить, а `newval` — новое значение, которое надо записать (0 или 1). Если $t = 3$, то это операция `bitsetAny`, и указано ещё два параметра `left` и `right` — отрезок, на котором нужно искать единичные биты. Если единичный бит на отрезке есть, то надо вывести `some`, а если нет — то `none`.

Размер массива `num` больше нуля и не превышает 10^7. Гарантируется, что после операции `bitsetZero` с параметром `num` все последующие операции обращаются только к битам с номерами от 0 до `num`-1 включительно — как минимум до следующего вызова `bitsetZero`.

Сумма значений `num` по всем операциям `bitsetZero` не превышает $2 \cdot 10^9$. Сумма длин отрезков (`right-left`) по всем операциям `bitsetAny` не превышает $2 \cdot 10^9$.

Формат выходных данных

Для каждой операции `bitsetGet` или `bitsetAny` нужно вывести ответ в отдельной строке.

Пример

input.txt	output.txt
14 0 100 2 30 1 2 31 1 2 32 1 1 31 1 7 2 31 0 1 31 3 30 33 3 31 32 3 0 100 3 45 67 0 48 1 30	1 0 0 some none some none 0

Пояснение к примеру

Сначала инициализируется массив из 100 битов. Потом устанавливаются в 1 биты с номерами 30, 31, 32. Потом делается запрос на значения битов 31 и 7 — они равны 1 и 0 соответственно. Далее 31-ый бит зануляется и запрашивается его уже нулевое значение. Потом делаются запросы о том, есть ли единичные биты на отрезках $[30, 33)$, $[31, 32)$ и $[0, 100)$. Обратите внимание, что правый конец отрезка в данной задаче исключается. Наконец, выполняется пересоздание массива, теперь размера 48 битов, в результате 30-ый бит становится нулевым.

Задача 3. XOR double

Источник:	базовая
Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	1 секунда
Ограничение по памяти:	разумное

Требуется взять заданное число X типа `double`, "прохогить" его с заданным 64-битным целым числом M, и вывести результат как `double`. "Прохогить" означает: обратить в битовом представлении X все биты, для которых бит с тем же номером в битовом представлении M равен единице.

В этой задаче предполагается little-endian порядок байтов и общепринятое 8-байтовое представление `double`.

Формат входных данных

В первой строке записано целое число N — количество тестов ($1 \leqslant N \leqslant 1\,000$). В остальных N строках записаны тесты, по одному в строке.

Каждый тест описан в формате: `"P/Q xor M"`. Здесь целые числа `P` и `Q` — числитель и знаменатель дроби, задающей вещественное число X ($0 \leqslant P \leqslant 100$, $1 \leqslant Q \leqslant 100$), а `M` — шестнадцатеричное целое число M ровно из шестнадцати цифр. В записи `M` сначала идут старшие цифры, потом младшие (как обычно у людей записываются числа).

Совет: Шестнадцитеричное число можно читать при помощи формата `"%x"` так же, как мы считаем десятичные числа форматом `"%d"`. Если нужно прочитать 64-битное число, то нужно дописать две буквы `ll` перед последней буквой формата.

Формат выходных данных

Для каждого теста выведите в отдельной строке (X xor M) как вещественное число типа `double`.

Ваши ответы должны быть верны с относительной точностью 10^{-14}. Рекомендуется использовать формат `"%0.15g"` при выводе ответа.

Пример

input.txt	output.txt
10	-0
0/1 xor 8000000000000000	1
1/1 xor 0000000000000000	-1
1/1 xor 8000000000000000	0
1/1 xor 3ff0000000000000	2
1/1 xor 7ff0000000000000	0.5
1/1 xor 0010000000000000	1.625
1/1 xor 000a000000000000	-0.428571428571429
3/7 xor 8000000000000000	-2.90689205178751e-054
3/7 xor 8b0abc0000000000	0.428571428570292
3/7 xor 000000000000d000	

Пояснение к примеру

В первом тесте $X = 0/1$, а в заданной маске M установлен только старший бит. В представлении нуля в `double` все биты нулевые, после операции xor старший бит становится

единичным, однако число по-прежнему остаётся нулевым (получается так называемый “отрицательный ноль”).

Во втором тесте число $X = 1/1$ равно единице, а маска M вся нулевая. Значит xor ничего не меняет и результат получается тоже равен единице.

В третьем и восьмом тестах в маске только старший бит единичный. Он в представлении `double` отвечает за знак числа, так что в этих тестах число X меняет знак.

В предпоследнем тесте число $X = 3/7$, его битовое представление выглядит как `3fdb6db6db6db6db` в шестнадцатеричном виде. Когда мы xor-им с заданной маской, получается представление `b4d1d1b6db6db6db`. Если проинтерпретировать эти данные как `double`, то получается число `-2.90689205178751e-054`.

Задача 4. Бинарный поиск

Источник:	основная
Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	1 секунда∗
Ограничение по памяти:	разумное

Требуется реализовать бинарный поиск элемента с заданным значением в отсортированном массиве.

В первой строке записано одно целое число N — размер отсортированного массива ($1 \leqslant N \leqslant 10^5$). Далее записаны элементы массива A_i (N целых чисел, $|A_i| \leqslant 10^9$). Затем записано целое число Q — количество запросов, которые нужно обработать ($1 \leqslant Q \leqslant 10^5$). В остальных Q строках записаны целые числа X_j, определяющие запросы на поиск.

Каждый запрос нужно обрабатывать следующим образом. Сначала нужно прибавить к записанному в файле числу X_j ответ на предыдущий запрос R_{j-1}, получив $Y_j = X_j + R_{j-1}$. Затем нужно найти в массиве A элемент, равный Y_j: его индекс будет ответом R_j для этого запроса. Если таких элементов много, то в качестве ответа R_j следует выбрать самый большой индекс. Если таких элементов нет, то ответ R_j равен -1.

Элементы массива нумеруются индексами от 0 до $N - 1$. Для первого запроса предыдущего ответа нет, так что полагаем $Y_0 = X_0$.

Пример

input.txt	output.txt
10 1 1 3 4 4 7 8 10 10 12 10 1 2 3 4 5 6 7 8 9 10	1 2 -1 2 5 -1 -1 5 -1 -1

Пояснение к примеру

Первый запрос: нужно найти значение $X_0 = Y_0 = 1$. Таких элементов два и они имеют индексы 0 и 1. В данной задаче нужно всегда выбирать максимальных индекс, если выбор есть, поэтому ответ A_0 равен 1.

Для следующего запроса задано число $X_1 = 2$. Прибавляем к нему предыдущий ответ $A_0 = 1$, и получаем число $Y_1 = 3$, которое нужно искать. Такой элемент есть в массиве под индексом 2, так что выводим ответ $A_1 = 2$.

Для следующего запроса указано $X_2 = 3$. Прибавляем предыдущий ответ $A_1 = 2$, и получаем, что нужно искать $Y_2 = 5$. Такого числа нет, так что выводим ответ $A_2 = -1$.

Теперь рассмотрим запрос $X_3 = 4$. Сперва прибавляем предыдущий ответ $A_2 = -1$, получаем число $Y_3 = 3$, которое нужно искать. Значит ответ $A_3 = 2$. И так далее...

Задача 5. Поиск ближайшего

Источник:	основная*
Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	1 секунда*
Ограничение по памяти:	разумное

Требуется реализовать поиск в отсортированном массиве элемента, ближайшего к заданному значению.

В первой строке записано одно целое число N — размер отсортированного массива ($1 \leqslant N \leqslant 10^5$). Далее записаны элементы массива A_i (N целых чисел, $|A_i| \leqslant 10^9$). Затем записано целое число Q — количество запросов, которые нужно обработать ($1 \leqslant Q \leqslant 10^5$). В остальных Q строках записаны целые числа Y_j, определяющие запросы на поиск.

Запросы нужно обрабатывать следующим образом. Для каждого числа Y_j нужно найти в массиве A такой элемент, чтобы разность между ним и Y_j была минимальной. В выходной файл нужно вывести два целых числа через пробел: индекс найденного элемента и полученное минимальное расстояние.

Элементы массива нумеруются индексами от 0 до $N - 1$. Если ближайших элементов несколько, разрешается выводить индекс любого из них.

Важно: В данной задаче ответ для каждого запроса нужно находить независимо от всех остальных запросов за время $O(\log N)$ в худшем случае.

Пример

input.txt	output.txt
10 1 1 3 4 4 7 8 10 10 12 10 1 2 3 4 5 6 7 8 9 10	0 0 2 1 2 0 3 0 4 1 5 1 5 0 6 0 7 1 7 0

Пояснение к примеру

В первом запросе нужно найти значение 1: в массиве два таких элемента, они имеют индексы 0 и 1. Расстояние получается нулевым.

Во втором запросе нужно искать число 2. Ближайшие к нему элементы массива — это число 1 (индексы 0 и 1) и число 3 (индекс 2). Соответственно первое число ответ может быть 0, 1 или 2, а второе (расстояние) должно быть равно единице.

Задача 6. Считалочка

Источник:	основная*
Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	1 секунда
Ограничение по памяти:	разумное

Дано два целых числа N и K, нужно промоделировать простую считалочку.

В кругу стоят N человек, пронумерованных числами от 1 до N в направлении по часовой стрелке. Начиная с первого человека, начинают считать по часовой стрелке, и каждый K-ый посчитанный выходит из круга. Так делают до тех пор, пока из круга не выйдут все кроме одного человека.

В выходной файл нужно вывести N целых чисел (по одному в строке) — номера людей в порядке их выбывания из круга, включая даже самого последнего.

Важно: в целях обучения, при решении данной задачи требуется хранить стоящих в кругу людей в связном списке любого вида.

Ограничения: $3 \leqslant N \leqslant 5\,000$, $2 \leqslant K < N$.

Пример

input.txt	output.txt
10 3	3 6 9 2 7 1 8 5 10 4

Пояснение к примеру

В кругу 10 человек, начинаем считать с 1-ого, и выбывает каждый третий. На первом круге вылетают 3-ый, 6-ой и 9-ый, остаются люди с номерами 1, 2, 4, 5, 7, 8, 10. На втором круге вылетают 2-ый и 7-ой, остаются номера 1, 4, 5, 8, 10. Дальше вылетают 1-ый и 8-ой, остаются номера 4, 5, 10. Наконец, выбывают 5-ый и 10-ый, а 4-ый остаётся последним.

Задача 7. Пересечение множеств

Источник:	основная
Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	4 секунды*
Ограничение по памяти:	разумное

Дан набор из N различных битовых масок. Две битовые маски называются конфликтующими, если у них есть общий единичный бит (хотя бы один). Иными словами, если k-ый бит равен единице в обеих масках для некоторого k, тогда маски конфликтуют. Требуется определить, сколькими способами можно выбрать пару неконфликтующих масок из заданного набора.

Формат входных данных

В первой строке входного файла задано целое число N — количество масок ($1 \leqslant N \leqslant 3 \cdot 10^4$). В каждой из оставшихся N строк записано по одной битовой маске. Битовая маска — это беззнаковое 64-битное число, заданное в шестнадцатеричном виде ровно шестнадцатью цифрами.

Для чтения битовых масок рекомендуется использовать формат: `%llx`

Формат выходных данных

Выведите одно целое число (в десятичном виде) — количество пар неконфликтующих масок.

Пример

input.txt	output.txt
7 0000000000000000 FFFFFFFFFFFFFFFF 0000100000000000 0000000004000000 0000A00000000000 0000666666600000 0000000012345000	13

Комментарий

Чтобы решение работало быстро, нужно определять наличие конфликта очень быстро, не перебирая отдельные биты в масках.

Задача 8. Аллокатор

Источник:	повышенной сложности*
Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	1 секунда
Ограничение по памяти:	**специальное**

Функции `malloc` и `free` позволяют динамически выделять блоки памяти и возвращать их назад в кучу. К сожалению, иногда эти функции работают медленнее, чем того хотелось бы. В таких случаях программисты порой реализуют собственные алгоритмы выделения памяти взамен `malloc`/`free`, которые применимы в одной конкретной задаче, зато работают при этом намного быстрее.

В этой задаче нужно реализовать специальный алгоритм для выделения блоков памяти **одинакового** (и маленького) размера. Алгоритм работает очень просто. Изначально выделяется большой кусок памяти (по сути массив) размером ровно в N блоков. Когда пользователь запрашивает у аллокатора новый блок памяти, аллокатор выбирает любой незанятый блок из этого массива и возвращает его адрес пользователю. Если все блоки заняты, аллокатор должен вернуть нулевой указатель, так как заведуемая им память закончилась. Когда пользователь освобождает блок памяти, аллокатор помечает его как свободный, чтобы в будущем можно было его переиспользовать.

Кроме собственно массива блоков, аллокатор также должен хранить множество незанятых блоков, чтобы знать, какие блоки сейчас можно выдавать пользователю, а какие нет. Для этого используется система под названием free list. Все незанятые блоки объединяются в односвязный список, причём узлами этого списка становятся сами незанятые блоки памяти. То есть в каждом незанятом блоке аллокатор хранит указатель на следующий такой незанятый блок.

Обратите внимание, что узлы односвязного списка **физически расположены внутри того самого массива**, блоки которого выдаются пользователю, а не где-то ещё снаружи! Так получается аллокатор без накладных расходов: помимо собственно куска памяти из N блоков не нужно никакой дополнительной памяти, кроме $O(1)$ памяти где-то в головной структуре аллокатора.

В тестовой задаче нужно реализовать аллокатор для выделения блоков размером в 8 байт и записи туда вещественных значений типа `double`. Нужно реализовать следующие функции:

```
//головная структура аллокатора
typedef struct MyDoubleHeap_s {
    ???     //можно хранить здесь всякие данные
} MyDoubleHeap;
//создать новый аллокатор с массивом на slotsCount блоков
MyDoubleHeap initAllocator(int slotsCount);
//запросить блок памяти под число типа double
double *allocDouble(MyDoubleHeap *heap);
//освободить блок памяти, на который смотрит заданный указатель
void freeDouble(MyDoubleHeap *heap, double *ptr);
```

Далее нужно обработать набор операций/запросов.

Формат входных данных

В первой строке задано два целых числа: N — на сколько блоков нужно изначально создать массив (`slotsCount`) и Q — сколько операций нужно после этого выполнить ($2 \leqslant N, Q \leqslant 3 \cdot 10^5$). В остальных Q строках описаны операции.

Каждая операция начинается с целого числа t — типа операции. Если $t = 0$, то это операция выделения блока памяти. Тогда далее записано вещественное число, которое нужно сохранить в этом блоке памяти. При выполнении этой операции нужно вывести в выходной файл адрес, который вернула функция `allocDouble`. Этот адрес должен должен делиться на 8, чтобы double был корректно выровнен по своему размеру.

Если $t = 1$, то это операция освобождения блока памяти, и далее записано целое число k — номер операции, в которой был выделен тот блок памяти, который сейчас нужно удалить. Если $t = 2$, то нужно просто распечатать содержимое того блока памяти, который был выделен на k-ой операции, как вещественное число.

Все запросы нумеруются по порядку номерами от 0 до $Q - 1$. Для вывода в файл адреса/указателя используйте формат `"%p"`. Все вещественные числа заданы с не более чем 5 знаками после десятичной точки, и не превышают 10^4 по модулю. Вещественные числа следует выводить в аналогичном виде (например, используя формат `"%0.5lf"`).

Гарантируется, что никакой выделенный блок памяти не будет удалён дважды, и что у вас не попросят распечатать содержимое уже освобождённого блока. Гарантируется, что если запрос на выделение памяти возвращает нулевой указатель (когда все N блоков заняты), то на эту операцию не ссылаются никакие другие запросы, т.е. этот невыделенный блок не попытаются освободить или распечатать.

Формат выходных данных

Выведите результаты выполнения операций (для операций типа $t = 0$ и $t = 2$).

Пример

input.txt	output.txt
5 12 0 0.1 0 1.1 0 2.1 0 3.1 0 4.1 0 5.1 2 3 2 1 1 3 0 123.0 2 9 0 -1	001A0480 001A0488 001A0490 001A0498 001A04A0 00000000 3.100000000000000 1.100000000000000 001A0498 123.000000000000000 00000000

Пояснение к примеру

Изначально вызываем `initAllocator` с `slotsCount` = 5. Внутри он создаёт массив на 5 элементов, и объединяет их все в односвязный список (т.к. изначально все блоки незаняты).

Далее делается шесть запросов на выделение памяти. Первые пять срабатывают успешно, и в выходном файле распечатаны адреса блоков (они идут подряд с шагом в 8 байт). Для шестого запроса свободных блоков нет (ведь $N = 5$), поэтому блок не выделяется и выводится нулевой указатель.

Далее распечатываются вещественные числа по адресам `001A0498` и `001A0488`. Потом блок по адресу `001A0498` освобождается, и сразу же выделяется обратно для числа 123.0. Наконец, распечатывается содержимое для только что выделенного блока (т.е. 123.0) и выполняется ещё один неуспешный запрос на выделение памяти.

Комментарий

Для решения тестовой задачи рекомендуется завести глобальный массив `double *idToHeap[301000]`, чтобы отслеживать, какой указатель `double*` соответствует каждому номеру операции k (см. формат входных данных).

Задача 9. XOR-список

Источник:	повышенной сложности*
Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	1 секунда
Ограничение по памяти:	разумное

В узле односвязного списка хранится один указатель, но по такому списку нельзя перемещаться в обратную сторону и неудобно удалять элементы. В двусвязном списке всё просто и красиво, но приходится хранить уже по два указателя в каждом узле. Данная задача посвящена очень странной конструкции, когда в каждом узле хранится только одно дополнительное значение, но при этом можно перемещаться в обе стороны.

Как известно, указатель — это просто адрес в памяти, а над адресами можно выполнять арифметические операции. В узле предлагаемого списка вместо указателей `next` и `prev` нужно хранить, к примеру, сумму этих двух адресов. Тогда если в программе известны указатели на два подряд идущих элемента такого списка `left` и `right`, то адрес соседнего узла можно найти вычитанием адреса одного узла из хранящейся в другом узле суммы адресов соседей. Общий принцип заключается в том, что для выполнения любой операции (даже перемещения) в таком списке необходимо иметь два указателя, которые смотрят на два подряд идущих элемента списка, потому что по одному указателю невозможно найти соседние узлы.

Канонически, вместо суммы адресов соседей в узле обычно хранят “побитовое исключающее или” (т.е. XOR) адресов соседей. Тому есть несколько причин: во-первых, операция XOR играет роль сложения и вычитания одновременно; во-вторых, операция XOR порой выполняется немного быстрее сложения и вычитания, т.к. её проще реализовать на аппаратном уровне. Канонический вариант XOR-списка описан в википедии.

Требуется реализовать XOR-список и решить тестовую задачу. Примерно как в задаче “Список с указателями”, но теперь в узлах надо хранить целочисленные значения.

Нужно реализовать следующие функции:

```
#define VALTYPE int      //тип значений: целое
typedef struct Node_s {
    size_t xorlinks;     //XOR адресов соседних узлов
    VALTYPE value;       //значение в этом узле
} Node;
typedef struct List_s {
    ???                  //тут можно хранить всякие данные
} List;

//инициализировать пустой список в list
void initList(List *list);
//добавить новый узел со значением val между соседними узлами left и right
Node *addBetween(Node *left, Node *right, VALTYPE val);
//дано два соседних узла left и right, нужно удалить узел left
VALTYPE eraseLeft(Node *left, Node *right);
//дано два соседних узла left и right, нужно удалить узел right
VALTYPE eraseRight(Node *left, Node *right);
```

Для простоты реализации рекомендуется сделать XOR-список кольцевым с **двумя** вспомогательными элементами (“начало” и “конец”).

Формат входных данных

В первой строке файла записано одно целое число T — количество тестов в файле. Далее в файле идут тесты (T штук) подряд, один за другим.

Первая строка теста начинается с целого числа Q — количество операций, которые нужно выполнить ($0 \leqslant Q \leqslant 10^5$). В начале каждого теста список пустой.

Затем идут Q строк, которые описывают операции над списком. В каждой строке сначала записан тип операции: 1 — удаление правого, -1 — удаление левого, 0 — добавление. Затем указано два индекса: для узла `left` и узла `right`. Если описывается операция вставки, то в конце также задано целочисленное значение нового узла.

Гарантируется, что указанный узел `right` всегда идёт сразу после указанного узла `left`. Один из этих узлов может быть не указан (тогда записан индекс -1), если выполняется операция на краю списка.

Все значения узлов лежат в диапазоне от 0 до 10^6 включительно.

Сумма Q по всем тестам не превышает 10^5.

Формат выходных данных

Для каждого теста нужно вывести строковые значения всех узлов списка после выполнения операций (в порядке их следования в списке), и строку `"==="` в конце.

Пример

input.txt	output.txt
2 5 0 -1 -1 4283 0 -1 0 2718 0 0 -1 5000 0 -1 1 1111 0 1 0 3141 7 0 -1 -1 0 0 -1 0 1000 0 0 -1 2000 0 1 0 3000 0 2 -1 4000 1 0 2 -1 0 4	1111 2718 3141 4283 5000 === 1000 3000 4000 ===

Задача 10. Битовый массив 2

Источник:	повышенной сложности
Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	1 секунда*
Ограничение по памяти:	**5 мегабайт**

Это расширенная версия задачи "Битовый массив". В дополнение к функциям из оригинальной задачи, нужно также реализовать массовое изменение значения на отрезке и вычисление количества единиц на отрезке.

Сигнатура дополнительных функций, которые надо реализовать:

```
//установить в val значение всех k-ых битов для left <= k < right
void bitsetSetSeg(bitword *arr, int left, int right, int newval);
//посчитать, сколько битов равно единице на отрезке left <= k < right
int bitsetCount(const bitword *arr, int left, int right);
```

При помощи реализованной структуры нужно решить тестовую задачу.

Внимание: в этой задаче ограничение по времени выставлено очень сурово!

Формат входных данных

В первой строке записано целое число N — количество операций, которые нужно обработать ($1 \leqslant N \leqslant 10^5$). В каждой из следующих N строк описывается одна операция над битовым массивом.

Описание операции начинается с целого числа t, обозначающего тип операции. Если $t = 0$, то это операция `bitsetZero`, и вторым целым числом в строке указан размер массива `num`. Если $t = 1$, то это операция `bitsetGet`, и в строке также записано целое число `idx` — номер бита. Значение этого бита нужно выдать в выходной файл. Если $t = 2$, то это операция `bitsetSet`, и в строке также содержатся целые числа `idx` и `newval`. Здесь `idx` — номер бита, который нужно изменить, а `newval` — новое значение, которое надо записать (0 или 1). Если $t = 3$, то это операция `bitsetSetSeg`, и указано ещё три параметра `left`, `right` и `newval` — отрезок, на котором нужно выполнить присваивание, и новое значение для битов этого отрезка. Если $t = 4$, то это операция `bitsetCount`, и указано ещё два параметра `left` и `right` — отрезок, на котором нужно посчитать количество единичных битов. Полученный результат нужно выдать в выходной файл.

Размер массива `num` больше нуля и не превышает 10^7. Гарантируется, что после операции `bitsetZero` с параметром `num` все последующие операции обращаются только к битам с номерами от 0 до `num`-1 включительно — как минимум до следующего вызова `bitsetZero`.

Сумма значений `num` по всем операциям `bitsetZero` не превышает $2 \cdot 10^9$. Сумма длин отрезков (`right-left`) по всем операциям `bitsetSetSeg` не превышает $2 \cdot 10^9$, аналогично для операций `bitsetCount`.

Формат выходных данных

Для каждой операции `bitsetGet` или `bitsetCount` нужно вывести ответ в отдельной строке.

Пример

input.txt	output.txt
10 0 100 3 17 54 1 4 10 25 1 17 1 54 1 32 2 31 0 3 15 20 0 4 10 20 4 19 59	8 1 0 1 0 33

Комментарий

Вычисление количества единичных битов в целом числе — весьма интересная задача сама по себе. Для решения этой задачи вам потребуется какое-нибудь быстрое решение. Много хороших вариантов можно найти в bit twiddling hacks (эта страница вообще очень познавательная) или в этом вопросе на SO.