

Задача 1. $A + B$

Источник:	базовая
Имя входного файла:	<code>input.bin</code>
Имя выходного файла:	<code>output.bin</code>
Ограничение по времени:	1 секунда
Ограничение по памяти:	разумное

Во входном файле дано восемь байт, которые задают два 32-битных знаковых целых числа: в первых четырёх байтах записано число A , а в последних четырёх — число B . Нужно вычислить полусумму этих чисел, округляя полученный результат **вниз**. Полученную полусумму вывести в выходной файл как 32-битное знаковое целое число. Все три числа заданы с порядком байтов `little-endian`.

Внимание: В качестве A и B могут быть даны любые числа из диапазона знаковых 32-битных целых чисел (т.е. от `INT_MIN` до `INT_MAX`). Настоятельно рекомендуется протестировать решение на числах, близких к крайним значениям, а также на разных комбинациях чётности чисел. Возможно, будет удобнее использовать 64-битные целые для промежуточных результатов, чтобы избежать переполнения.

Весь ввод и вывод в данной задаче бинарный. Ниже показано hex-представление бинарных данных: каждая группа из двух цифр обозначает один байт в файле. Вашей программе на вход будет подан файл с 8 байтами данных, и программа должна создать файл с 4 байтами ответа. Для создания, редактирования и просмотра бинарных файлов используйте какой-нибудь Hex-редактор, например HxD. Пример входных и выходных данных в бинарном виде можно скачать по ссылке.

Примеры

<code>input.bin</code>	<code>output.bin</code>
AB 05 00 00 12 30 00 00	DE 1A 00 00
FF FF FF FF FE FF FF FF	FE FF FF FF
0A 0D 0A 0D 0D 0A 0D 0A	8B 8B 8B 0B
00 00 00 80 00 00 00 80	00 00 00 80

Пояснение к примеру

В первом примере даны числа $A = 1451$ и $B = 12306$. Сумма равна 13757, после деления на два получаем 6878.

Во втором примере даны числа $A = -1$ и $B = -2$. Сумма равна -3 , при делении на два получается -2 (округление вниз).

В третьем примере даны большие положительные числа. Если не работает, убедитесь, что открываете файлы в бинарном режиме.

В четвертом примере числа одинаковы и равны `INT_MIN` $= -2^{31}$. Очевидно, полусумма также равна `INT_MIN`.

Задача 2. Сумма чисел

Источник:	базовая
Имя входного файла:	<code>input.bin</code>
Имя выходного файла:	<code>output.bin</code>
Ограничение по времени:	1 секунда
Ограничение по памяти:	разумное

В первых четырёх байтах входного файла задано число N — количество чисел в последовательности ($1 \leq N \leq 10\,000$). Далее заданы сами целые числа последовательности: N штук по 4 байта в каждом. Все числа по абсолютной величине не превышают 10^5 .

Требуется найти сумму N чисел последовательности и вывести её как 4-байтовое целое число.

Внимание: Весь ввод и вывод в данной задаче бинарный. В каждом конкретном тесте все числа записаны с одинаковым порядком байтов: это может быть либо **big-endian**, либо **little-endian**. Однако в разных тестах порядок байтов может быть разным. Выводить число в ответ требуется с тем же порядком байтов, с которым заданы входные данные.

Пример

<code>input.bin</code>
00 00 00 05 00 00 02 A7 00 00 00 A0 00 00 03 CD 00 00 00 78 00 00 01 B8
<code>output.bin</code>
00 00 09 44

Пояснение к примеру

Учтите, что в примере указано лишь hex-представление бинарных данных! Пример входных и выходных данных можно скачать по ссылке.

Задача 3. Файлы

Источник:	базовая
Имя входного файла:	<code>input.bin</code>
Имя выходного файла:	<code>output.bin</code>
Ограничение по времени:	1 секунда
Ограничение по памяти:	разумное

Вася хочет найти кое-какой интересный файл в списке, но в сожалению не помнит его имени. Он только помнит, что создал этот файл не раньше, чем в момент времени A , и что после момента времени B он точно его **не** менял. Кроме того, он абсолютно уверен, что этот файл был ви́ден при просмотре. Нужно написать программу, которая будет отбирать файлы, подходящие под этот критерий, и выдавать их Васе.

Список файлов и директорий задан в бинарном виде. Все целые числа записаны с little-endian порядком байтов. Любой момент времени (timestamp) задаётся как количество 100-наносекундных интервалов, прошедших с 1 января 1601 года (текущий момент времени примерно равен 131 832 294 671 670 965). Далее и файлы, и директории мы будем называть обобщённо «файлами».

Входные данные заданы в следующем формате (слева указан диапазон байтов):

0-3	целое число N — количество файлов в списке ($1 \leq N \leq 1\,000$)
4-11	целое число A — искомый файл был создан не раньше этого момента времени ($0 < A < 10^{18}$)
12-19	целое число B — искомый файл не модифицировался позже этого момента времени ($A < B < 10^{18}$)
20-??	В оставшейся части входных данных подряд идут описания N файлов.

Каждый отдельный файл описывается непрерывным блоком данных в формате:

0-20	Имя файла, состоящее из латинских букв, цифр и символов точки. Длина имени не превышает 20, все оставшиеся байты заполнены нулями.
21-28	Размер S_i файла в байтах ($0 \leq S_i \leq 10^{12}$).
29	Байт равен 1, если файл на самом деле является директорией, и 0 иначе.
30-37	Момент времени C_i , когда файл был создан ($0 < C_i < 10^{18}$).
38-45	Момент времени M_i , когда файл был изменён в последний раз ($C_i < M_i < 10^{18}$).
46	Байт равен 1, если файл скрытый, и 0 иначе.

Гарантируется, что все заданные файлы имеют разные имена. Нужно выбрать среди них файлы, удовлетворяющие критериям:

1. Не директория и не скрытый.
2. Создан не раньше времени A , последний раз изменён не позже времени B .

Эти файлы нужно отсортировать по имени в лексикографическом порядке, после чего вывести их все подряд в выходные данные. Выводить каждый файл следует блоком данных в том же формате, в котором он был задан во входных данных. Учтите, что при лексикографическом сравнении символы сравниваются по их ASCII-коду.

Замечание: Чтобы не писать много кода, можно читать и писать файл целиком как структуру.

Пример

Пример входных и выходных данных можно скачать по ссылке.

Задача 4. Простой BSON

Источник:	основная
Имя входного файла:	<code>input.bin</code>
Имя выходного файла:	<code>output.txt</code>
Ограничение по времени:	1 секунда
Ограничение по памяти:	разумное

Формат BSON является бинарным вариантом известного формата JSON. Его спецификация доступна по адресу: <http://bsonspec.org/spec.html>

В файле `input.bin` записан документ (объект) в формате BSON, внутри которого нет поддокументов (подобъектов и подмассивов). В этом объекте могут быть лишь поля следующих типов: `double`, `string`, `bool`, `null`, `int32`, `int64`. Нужно прочитать этот BSON, и вывести его в текстовом виде как JSON.

В описании формата BSON используются следующие базовые типы:

- `byte`: просто байт.
- `int32`: знаковое 32-битное целое число.
- `int64`: знаковое 64-битное целое число.
- `double`: вещественное число двойной точности.

Все числа записываются с `little-endian` порядком байтов. Целые числа записываются в дополнительном коде, а вещественные — согласно стандарту IEEE 754.

Документ BSON начинается с числа типа `int32`, которое обозначает размер всего документа в байтах. Далее идёт описание всех полей документа одно за другим. В самом конце документа стоит дополнительный нулевой байт.

Описание поля начинается с одного байта T , который определяет, какого типа значение в нём записано:

- $T = 1$: значение типа `double`.
- $T = 2$: значение типа `string`.
- $T = 8$: значение типа `bool`.
- $T = 10$: значение типа `null`.
- $T = 16$: значение типа `int32`.
- $T = 18$: значение типа `int64`.

Сразу после типа записано имя поля в виде строки, заканчивающейся на дополнительный нулевой байт. После этого нулевого байта записано значение поля.

Значение поля типа `bool` может быть равно либо `false`, либо `true`. Значение `false` записывается в BSON-файле как байт с нулевым значением, а `true` — как байт с единичным. Для поля типа `null` никаких байтов в значение поля **не** пишется.

Значение поля типа `string` записывается следующим образом. Сначала записано число $L > 0$ типа `int32`, которое равно количеству байтов в строке, включая дополнительный нулевой байт. В следующих $L - 1$ байтах задана сама строка. В конце добавлен дополнительный нулевой байт.

При записи документа в текстовом формате JSON в первой строке пишется открывающая фигурная скобка, а в последней — закрывающая фигурная скобка. Каждая строка между ними описывает одно поле. Поля нужно описывать в том же порядке, в котором они даны в BSON-файле. Описание поля начинается с имени поля, заключённого в двойные кавычки, и двоеточия сразу после него. Далее должен быть поставлен один пробел, после которого записано значение поля. В самом конце строки должна стоять запятая, если только это поле не является последним полем документа.

Значение типа `string` заключается в двойные кавычки, остальные значения — нет. Зна-

чение типа `bool` пишется как одно из слов `false` или `true`. Значение типа `null` пишется как слово `null`. Значение целочисленного типа пишется в десятичной системе исчисления без ведущих нулей (как обычно). Значение типа `double` должно быть записано при помощи `printf` с форматом `"%0.15g"`.

Гарантируется, что все числа типа `double` являются нормальными числами и по модулю не превышают 10^{100} . Гарантируется, что все строковые значения и имена полей **не** содержат символов, требующих экранирования в формате JSON. То есть они не содержат символов обратного слэша (ASCII 92), двойных кавычек (ASCII 34), а также контрольных символов (ASCII 0-31). Учтите, что строковые значения и имена полей задаются в кодировке UTF-8, что в частности имеет значение для русских символов в примере.

Размер входного файла не превышает один килобайт.

Пример

input.bin
5F 00 00 00 02 48 57 00 0E 00 00 00 48 65 6C 6C
6F 2C 20 57 6F 72 6C 64 21 00 02 D0 9F 20 D0 9C
00 17 00 00 00 D0 9F D1 80 D0 B8 D0 B2 D0 B5 D1
82 20 D0 B2 D1 81 D0 B5 D0 BC 21 00 01 64 62 6C
00 66 66 66 66 66 BA 81 40 10 69 6E 74 00 25 00
00 00 0A 6E 6F 6E 65 00 08 62 6C 6C 00 01 00
output.txt
{
"HW": "Hello, World!",
"П М": "Привет всем!",
"dbl": 567.3,
"int": 37,
"none": null,
"bll": true
}

Комментарий

Пример входных и выходных данных можно скачать по ссылке.

В целях тестирования вы можете создавать BSON-файлы при помощи online-конвертера: <https://json-bson-converter.appspot.com> Учтите, что конвертация обратно в JSON на этом сайте **не** работает.

Кроме того, вы можете использовать для конвертации следующие скрипты на языке Python (не забудьте запустить “`pip install bson`” после установки Python 3):

```
import bson, json
with open("input.txt", "rt", encoding = "utf-8") as f:
    data = json.load(f)
with open("output.bin", "wb") as f:
    f.write(bson.dumps(data))
```

```
import bson, json
with open("input.bin", "rb") as f:
    data = bson.loads(f.read())
with open("output.txt", "wt", encoding = "utf-8") as f:
    json.dump(data, f, indent = 4, ensure_ascii = False)
    print("", file=f)
```

Задача 5. Побитовый вывод

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

Есть алфавит (набор) из N различных символов, пронумерованных числами от 0 до $N-1$. Есть текст из M символов, все символы которого выбраны из этого набора. Для каждого из N символов задано, какой последовательностью битов он кодируется. Нужно перекодировать весь текст в битовый массив и записать битовый массив в файл.

В данной задаче все числа записаны с little-endian порядком байтов. Более того, при записи битового массива мы считаем, что биты внутри байта также идут в little-endian порядке.

Формат входных данных

В первых 4 байтах входного файла дано целое число N — количество различных символов в алфавите ($1 \leq N \leq 10^4$). Далее для каждого из этих символов идёт описание битовой последовательности, на которую его надо заменять. Описание начинается с целого 4-байтового числа L — количества битов в последовательности ($1 \leq L \leq 64$). Далее идёт L байтов, каждый байт принимает значение 0 или 1 и по сути задаёт один бит в последовательности.

Описания битовых последовательностей даны в порядке увеличения номера символа. Биты в каждой последовательности даны в том порядке, в котором их надо записывать в выходной битовый массив.

Далее в файле записано 4-байтовое целое число M — длина текста в символах ($1 \leq M \leq 10^6$). Наконец, в конце файла записан текст как M символов, каждый записан как 2-байтовое целое число. Гарантируется, что все символы лежат в диапазоне от 0 до $N-1$.

Формат выходных данных

Поскольку записать в файл можно только целое количество байтов, битовый массив дополняется минимальным количеством нулевых битов так, чтобы его длина делилась на 8.

Пример

input.bin															
04	00	00	00	0D	00	00	00	00	01	01	00	01	00	01	01
00	00	01	01	01	0D	00	00	00	00	01	01	01	01	01	00
00	01	01	00	00	01	08	00	00	00	01	01	01	00	01	00
00	01	06	00	00	00	01	01	01	01	00	01	08	00	00	00
03	00	01	00	00	00	01	00	00	00	01	00	00	00	01	00
output.bin															
AF	CF	5C	E3	3E	73	8D	FB	CC	35	EE	33	01			

Пояснение к примеру

Пример в бинарном виде можно скачать по ссылке.

В алфавите 4 символа. Символ 0 заменяется на 1101011000111, символ 1 на 0111110011001, символ 2 на 11101001, а символ 3 на 111101. В тексте 8 символов: 3, 1, 0, 1, 0, 1, 0, 1.

В результате кодирования получается битовый массив: 11110101 11110011 00111010 11000111 01111100 11001110 10110001 11011111 00110011 10101100 01110111 11001100 1. В последней группе только один единичный бит, он дополняется нулями до единичного байта.

Задача 6. Кодировка UTF-8

Источник:	основная
Имя входного файла:	<code>input.bin</code>
Имя выходного файла:	<code>output.bin</code>
Ограничение по времени:	1 секунда
Ограничение по памяти:	разумное

Сегодня повсеместно используется представление символов различных языков при помощи таблицы Unicode. Самая популярная кодировка (т.е. способ сохранения в файл) для Unicode символов — это UTF-8. В данной задаче вам нужно прочитать из входного файла текст в кодировке UTF-8, преобразовать его в последовательность кодов символов, и сохранить все эти коды как 32-битовые целые числа в выходной файл (т.е. по сути записать в кодировке UTF-32LE).

Каждому символу в таблице Unicode назначен код — какое-то неотрицательное целое число. Коды символов таблицы Unicode покрывают весь диапазон от `0x0000` (нуля) до `0x10FFFF` включительно, за исключением диапазона от `0xD800` до `0xDFFF`, в котором расположены так называемые суррогаты, не соответствующие никаким символам. Таким образом, всего в таблице Unicode 1112064 (`0x10F800`) символов.

Кодировку UTF-8 можно использовать для представления произвольной последовательности беззнаковых целых чисел (не более 21 бита каждое) в виде последовательности байтов. Когда в UTF-8 кодируют текст Unicode, то в качестве этих беззнаковых чисел выступают коды символов текста. Чтобы закодировать последовательность значений в UTF-8, нужно представить каждое значение в виде последовательности от одного до четырёх байтов. Далее нужно записать все полученные последовательности байтов подряд друг за другом в порядке записи значений в исходной последовательности. Каждое отдельное значение кодируется согласно таблице:

длина	шаблон	битов
1	<code>0xxxxxxx</code>	7
2	<code>110xxxxx 10xxxxxx</code>	11
3	<code>1110xxxx 10xxxxxx 10xxxxxx</code>	16
4	<code>11110xxx 10xxxxxx 10xxxxxx 10xxxxxx</code>	21

Здесь в первом столбце записано количество байтов в представлении, во втором столбце — шаблон, а в третьем — количество букв ‘x’ в шаблоне. В шаблоне записаны в двоичном виде байты, которые получаются при его использовании для представления значения: цифры ‘1’ и ‘0’ обозначают фиксированные значения битов, а буква ‘x’ обозначает бит, состояние которого определяется по кодируемому числу.

Чтобы закодировать целое число V в виде последовательности из k байтов, нужно:

1. Записать число V в двоичной системе исчисления.
2. Найти в таблице k -ую строку и зафиксировать её для дальнейших шагов.
3. Если требуется, дополнить V ведущими нулями так, чтобы количество битов в V совпало с числом в последнем столбце таблицы.
4. В буквы ‘x’ шаблона вставить биты числа V . Биты вставляются слева направо, в порядке от старшего бита к младшему.
5. В результате вместо шаблона получилось k байтов, по восемь бит в каждом из них — результат кодирования V .

Заметим, что если число V в двоичном виде имеет больше битов, чем букв 'х' в шаблоне длины k , то закодировать его в k байтов нельзя. С другой стороны, для конкретного значения V вполне можно выбрать количество байтов k несколькими способами. Стандарт UTF-8 предписывает выбирать **минимально** возможное k . Если число закодировано в неминимальное количество байтов, то это называется "overlong encoding", и считается ошибкой.

Например, буква 'с' русского алфавита имеет код 1089 в таблице Unicode, что в двоичном виде выглядит как 10001000001. В UTF-8 это число можно записать тремя способами, при этом только первый способ правильный, а все остальные ошибочные:

- 11010001 10000001
- 11100000 10010001 10000001
- 11110000 10000000 10010001 10000001

Во входном файле записан текст в формате UTF-8, возможно с ошибками, общим размером не более мегабайта. Если ошибок нет, то ваша программа должна извлечь коды записанных символов и записать их в выходной файл как 4-байтовые целые числа с little-endian порядком байтов. Если в файле есть ошибки, то они должны быть обработаны строго определённым образом.

Ваша программа должна декодировать символы в том порядке, в котором они записаны в файле. Для каждого символа следует сперва обращать внимание на первый байт и определять по нему количество байтов в представлении символа. Далее нужно считывать остальные байты символа (они называются байтами продолжения) и получать код как целое число. Наконец, нужно проверять, что этот код есть в таблице Unicode и что не имеет место "overlong encoding".

Есть четыре вида жёстких ошибок:

1. Встретился байт, у которого пять или больше старших битов единичные.
2. Нужно прочесть байт продолжения, чтобы закончить символ, а в файле нет больше байтов.
3. Нужно прочесть байт продолжения, чтобы закончить символ, а следующий байт не имеет формат 10xxxxxx.
4. Первый байт очередного символа имеет формат 10xxxxxx.

При обнаружении жёсткой ошибки программа должна сразу завершаться, не читая остальное содержимое файла. При этом все предыдущие полностью прочитанные символы должны быть в выходном файле.

Кроме того, есть три вида мягких ошибок:

1. Записанный код символа больше 0x10FFFF, а значит не попадает в таблицу Unicode.
2. Записанный код символа попадает в диапазон от 0xD800 до 0xDFFF, то есть является суррогатом.
3. Записанный код символа можно было закодировать меньшим количеством байтов k , то есть он закодирован в виде "overlong encoding".

При обнаружении ошибки код символа следует заменить на 0xFFFD (так называемый "replacement char"), выдать его в выходной файл как обычно и продолжить работу над оставшейся частью файла.

Пример

На следующей странице приведены примеры в шестнадцатеричном виде. Скачать их в бинарном виде можно по ссылке. Рекомендуется также смотреть на примеры, записывая байты в двоичном представлении.

input.bin	
78 E2 89 A4 28 CE B1 2B CE B2 29 C2 B2 CE B3 C2 B2	
output.bin	
78 00 00 00 64 22 00 00 28 00 00 00 B1 03 00 00 2B 00 00 00	
B2 03 00 00 29 00 00 00 B2 00 00 00 B3 03 00 00 B2 00 00 00	
input.bin	
78 F7 89 A4 80 28 CE B1 2B CE B2 29 C2 B2 CE B3 C2	
output.bin	
78 00 00 00 FD FF 00 00 28 00 00 00 B1 03 00 00 2B 00 00 00	
B2 03 00 00 29 00 00 00 B2 00 00 00 B3 03 00 00	
input.bin	
63 6F 6F 6C 20 6D 61 74 68 3A C0 A0 78 E2 89 A4 28 CE B1 2B CE B2 29 C2 B2 CE B3 C2 20 78 79	
output.bin	
63 00 00 00 6F 00 00 00 6F 00 00 00 6C 00 00 00 20 00 00 00	
6D 00 00 00 61 00 00 00 74 00 00 00 68 00 00 00 3A 00 00 00	
FD FF 00 00 78 00 00 00 64 22 00 00 28 00 00 00 B1 03 00 00	
2B 00 00 00 B2 03 00 00 29 00 00 00 B2 00 00 00 B3 03 00 00	
input.bin	
F0 9F 98 82 78 ED A0 90 28 ED B4 80 CE B2 29 FE C2 B2 CE B3 C2 B2	
output.bin	
02 F6 01 00 78 00 00 00 FD FF 00 00 28 00 00 00 FD FF 00 00	
B2 03 00 00 29 00 00 00	
input.bin	
D1 82 D0 B5 D1 81 D1 82 20 85 D1 0D 0A	
output.bin	
42 04 00 00 35 04 00 00 41 04 00 00 42 04 00 00 20 00 00 00	

Пояснение к примеру

В первом тесте записан полностью правильный текст из десяти символов. В остальных трёх тестах есть как мягкие, так и жёсткие ошибки.

На втором тесте второй символ имеет код больше 0x10FFFF (мягкая ошибка), так что он заменяется на replacement char. В конце файла записан байт 0xC2, который является началом двухбайтового кода, однако байтов продолжения нет (файл закончился). Это жёсткая ошибка. При этом все символы, прочитанные до этого, сохранены в файл.

На третьем тесте сначала идёт много однобайтовых символов. В середине стоит символ пробела (код 32), записанный двумя байтами. Это overlong encoding, так как можно закодировать его одним байтом 0x20 — заменяется на replacement char (мягкая ошибка). Кроме того, почти в самом конце символ начинается байтом 0xC2, подразумевающим двухбайтовый код, но следующий байт 0x20 не является байтом продолжения — это жёсткая ошибка.

В четвёртом тесте третий и пятый символы имеют коды 0xD810 и 0xDD00 соответственно, которые являются суррогатами (мягкие ошибки). Ближе к концу обнаруживается байт 0xFE, который является жёсткой ошибкой, ибо в UTF-8 байт может иметь максимум четыре старших единичных бита. В пятом тесте после пяти символов внезапно стоит байт продолжения 0x85, хотя на этом месте должен быть первый байт символа — жёсткая ошибка.

Задача 7. Список по указателям

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

В данной задаче нужно реализовать набор функций для работы со связным списком, так чтобы пользователь оперировал напрямую указателями на лежащие внутри узлов значения. Список должен быть двусвязный, кольцевой со вспомогательным узлом. Каждый узел списка должен быть размещён в динамической памяти, выделенной с помощью `malloc`.

Нужно реализовать следующие функции:

```
typedef struct Node {
    struct Node *prev, *next;
    double value;
} Node;
//создаёт пустой список
//возвращает указатель на value в голове списка (вспом. узел)
double *initList();
//полностью освобождает память списка
//принимает указатель на value голове списка (вспом. узел)
void freeList(double *head);
//определяет следующий элемент после заданного
//(для вспом. узла возвращает первый элемент)
double *getNext(double* curr);
//определяет предыдущий элемент перед заданным
//(для вспом. узла возвращает последний элемент)
double *getPrev(double* curr);
//создаёт и вставляет узел со значением newval сразу после узла where
double *addAfter(double* where, double newval);
//создаёт и вставляет узел со значением newval сразу перед узлом where
double *addBefore(double* where, double newval);
//удаляет и освобождает заданный узел
void erase(double* what);
```

Для реализации этих функций нужно уметь находить указатель на узел по указателю на значение `value` внутри него.

Используя эти функции, решите тестовую задачу, аналогичную задаче “Список с указателями” (см. набор задач по теме связных списков).

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 ($0 \leq Q \leq 10^5$).

Затем идут Q строк, которые описывают операции над списком. В каждой строке сначала записан тип операции: 1 — добавление спереди, -1 — добавление сзади, 0 — удаление. Затем указан индекс узла. Если описывается операция вставки, то в конце также задано вещественное значение нового узла.

Все значения узлов заданы максимум с тремя знаками после десятичной точки, и по модулю не превышают 10^4 .

Сумма Q по всем тестам не превышает 10^5 .

Формат выходных данных

Для каждого теста:

1. При выполнении операции удаления узла выводите значение удаляемого узла
2. После выполнения всех операций выведите строку ===
3. Затем выведите значения всех оставшихся узлов списка в порядке их следования
4. Наконец, выведите ещё раз строку ===

Все вещественные числа следует выводить с помощью формата "%.31f"

Пример

input.txt	output.txt
2	===
5	1111.000
1 -1 4283	2718.000
-1 0 2718	3141.000
1 0 5000	4283.000
-1 1 1111	5000.000
1 1 3141	===
9	2000.000
1 -1 0	1000.000
1 -1 1000	3000.000
-1 -1 2000	===
1 1 3000	4000.000
-1 0 4000	0.000
0 2	9876.543
1 0 9876.543	===
0 1	
0 3	

Задача 8. Выравнивание структур

Источник:	повышенной сложности
Имя входного файла:	<code>input.txt</code>
Имя выходного файла:	<code>output.txt</code>
Ограничение по времени:	1 секунда
Ограничение по памяти:	разумное

Петя изучает язык C, и только что дошёл до структур. Его особенно заинтересовал тот факт, что поля структуры не всегда размещаются в памяти подряд: иногда между соседними полями появляются дополнительные «пустоты» (padding). Из-за этого получается, что размер структуры может зависеть от того, в каком порядке программист перечислит поля структуры! Очень интересна в этом плане, например, вот эта статья.

Дано описание структуры на языке C. В первой строке написано ключевое слово `struct` и открывающая фигурная скобка. В последней строке записана закрывающая фигурная скобка и точка с запятой. Каждая строка между ними описывает ровно одно поле структуры.

В описании поля сначала стоит тип поля, затем имя поля, и наконец точка с запятой. Тип поля отделён от имени поля хотя бы одним пробелом. Имя поля — это непустая строка, состоящая из латинских букв любого регистра, цифр и символов подчёркивания, причём имя точно **не** начинается с цифры.

В качестве типа поля может быть указан либо примитивный тип, либо указатель на примитивный тип. Во входных данных могут быть только следующие примитивные типы: `char`, `short`, `int`, `long`, `float`, `int64_t` и `double`. Указатель может быть любого порядка (т.е. двойной указатель, тройной указатель и т.п.).

В данной задаче будем считать, что структура размещается в памяти следующим образом (в реальности правила размещения определяются компилятором). Первое поле структуры помещается по адресу с максимальным выравниванием: будем считать, что его адрес делится нацело на 8. Каждое следующее поле размещается по такому адресу, что:

1. оно расположено после предыдущего,
2. оно корректно выровнено, то есть его адрес делится на его размер, и
3. его адрес минимален при выполнении первых двух условий.

При этом между соседними полями может появиться пустое место (padding) размером от 1 до 7 байт.

После последнего поля также может быть добавлено несколько байт пустоты. Это необходимо, чтобы можно было хранить массив таких структур, то есть располагать в памяти много одинаковых структур друг за другом. В конце структуры добавляется минимальное количество байт пустоты, так чтобы в массиве любой длины все поля всех структур были корректно выровнены. При этом размер всей структуры равен сумме размеров всех полей и размеров всех имеющихся в структуре пустот.

Петя может переставлять местами поля структуры. Он хочет узнать, какой при этом может получиться минимальный размер структуры, и какой максимальный размер. Как известно, размер также зависит от модели данных. Петя хочет узнать ответ для всех популярных моделей.

Формат входных данных

Во входном файле описана одна структура согласно указанным в условии правилам. В любом месте описания может быть добавлено любое количество пробелов (ASCII 32), если их добавление не разрывает на части название примитивного типа, имя поля или ключевое слово.

Гарантируется, что в структуре от 1 до 100 полей, а общее количество символов в описании не превышает 5 000. Имена всех полей структуры отличаются друг от друга и не совпадают с ключевыми словами языка C.

Формат выходных данных

В выходном файле должно быть четыре строки, по одной строке на модель данных. В каждой строке нужно записать через пробел два целых числа: минимально возможный размер структуры и максимально возможный размер.

В первой строке должен быть ответ для модели данных LP32, во второй — для модели ILP32, в третьей — для модели LLP64 и в четвёртой — для модели LP64. Информацию о том, сколько байтов занимает каждый тип на каждой модели данных, можно найти на странице: <https://ru.cppreference.com/w/cpp/language/types>

Пример

input.txt	output.txt
<pre>struct { int x; int64_t* * Y; char _temp1; } ;</pre>	<pre>8 12 12 12 16 24 16 24</pre>

Пояснение к примеру

Рассмотрим структуру в модели принятой на Win64 модели данных LLP64.

Поле `x` имеет размер 4 байта, поле `Y` размера 8 байт, а поле `_temp1` занимает 1 байт.

Если не менять порядок полей, тогда поле `x` будет занимать байты 0-3. Придётся добавить 4 байта пустоты, чтобы поле `Y` было выровнено по 8 байтам, заняв байты 8-15. Поле `_temp1` будет иметь адрес 16 и занимать один байт, но после него нужно добавить ещё 7 байтов пустоты. Без этого поле `Y` второй структуры в массиве не может быть корректно выровнено. Получается общий размер 24 байта.

Если выбирать порядок полей, то размер 24 байта получается только когда поле `Y` находится между другими двумя полями. Так получается в 2 способах среди всех 6 способов выбора порядка. В остальных случаях размер структуры равен 16 байтам.

Задача 9. TGA в BMP

Источник:	повышенной сложности
Имя входного файла:	<code>input.tga</code>
Имя выходного файла:	<code>output.bmp</code>
Ограничение по времени:	1 секунда
Ограничение по памяти:	разумное

В данной задаче требуется прочитать из файла растровое изображение в формате Targa (Truevision TGA) и записать это изображение в выходной файл в формате BMP.

О формате TGA можно прочитать в википедии. Более точно описание формате приведено здесь. В данной задаче про входной TGA гарантируется, что:

1. Байт 1 "Color Map Type" равен нулю, то есть файл не содержит карты цветов.
2. Байт 2 "Image Type" равен 2 или 10, то есть в файле хранится TrueColor изображение, причём возможно с RLE-сжатием, а возможно без него.
3. Байт 16 "Image Pixel Size" равен 24 или 32, то есть закодированы красная, зелёная и синяя компоненты, а иногда ещё и альфа-канал.
4. Файл может быть записан в новом формате, т.е. содержать подпись XTRUEVISION-XFILE в конце. Однако даже если это так, то следует полностью игнорировать опциональные области: "Developer Area", "Extension Area", "TGA File Footer".
5. Ширина и высота изображения положительные и не превышают 2048.

О формате BMP можно прочитать в википедии. На выходной файл накладываются следующие дополнительные требования:

1. Структура с информационными полями "DIB header" должна быть самой старой версии CORE и иметь размер 12 байт.
2. Поле `bcBitCount` должно быть равно 24, то есть пиксели должны задаваться в формате RGB (если был альфа-канал, то его надо выбросить).
3. Пиксельные данные должны начинаться сразу после всех хедеров, то есть поле `bf0ffBits` должно быть минимально возможным.

С учётом этих дополнительных правил существует единственный вариант выходного файла. Выданный вашим решением BMP-файл будет засчитан, только если он полностью побайтово совпадает с правильным ответом.

Пример

Пример входных и выходных данных можно скачать по ссылке.