

## Задача 1. Хранение строк

Источник:	базовая*
Имя входного файла:	<code>input.txt</code>
Имя выходного файла:	<code>output.txt</code>
Ограничение по времени:	1 секунда
Ограничение по памяти:	<b>специальное</b>

Нужно реализовать систему хранения строк в динамической памяти.

Система должна обрабатывать три вида запросов/операций:

0. *Создать строку*: указывается длина создаваемой строки и собственно содержимое строки. Нужно выделить блок памяти при помощи `malloc` и записать в него содержимое.
1. *Удалить строку*: указывается идентификатор ранее созданной строки. Нужно освободить память, выделенную ранее для этой строки, при помощи `free`.
2. *Вывести строку*: указывается идентификатор ранее созданной строки. Нужно распечатать содержимое этой строки в выходной файл.
3. *Сколько символов в строке*: указывается идентификатор ранее созданной строки и один символ. Нужно вывести в выходной файл одно целое число: сколько раз этот символ встречается в указанной строке.

Идентификатором строки является номер запроса на её создание в общей нумерации запросов. При обращении к строке по идентификатору гарантируется, что эта строка ещё **не** была удалена.

**Замечание:** Не все созданные строки удаляются в запросах. В целях аккуратности удалите все остающиеся строки при помощи `free` в конце программы.

### Формат входных данных

В первой строке записано целое число  $N$  — количество запросов ( $1 \leq N \leq 10^5$ ). В остальных  $N$  строках описаны запросы, по одному запросу в строке.

Запрос начинается с целого числа  $t$ , обозначающего тип запроса. Если  $t = 0$ , то это запрос создания, и тогда далее указана длина строки  $l$  и сама строка ( $1 \leq l \leq 10^5$ ). Если  $t = 1$ , то это запрос удаления, а если  $t = 2$  — то это запрос на вывод. В обоих случаях далее указано целое число  $k$  — идентификатор строки ( $0 \leq k < N$ ). Если  $t = 3$ , то это запрос о количестве символов, и тогда далее указан идентификатор строки  $k$  и ещё один символ через пробел.

Все строки состоят из произвольных печатаемых символов ASCII (коды от 33 до 126 включительно). То же верно и про символы в запросах на количество.

Сумма всех длин  $l$  по всем запросам создания не превышает  $5 \cdot 10^5$ . Суммарный размер всех строк, которые вам нужно вывести, не превышает  $5 \cdot 10^5$ . Сумма длин строк по всем запросам о количестве символов не превышает  $10^8$ .

## Пример

input.txt	output.txt
12	aba
0 3 aba	2
2 0	1
3 0 a	malloc
3 0 b	%d%s%
1 0	3
0 6 malloc	%d%s%
0 5 %d%s%	
2 1	
2 2	
1 1	
3 2 %	
2 2	

## Пояснение к примеру

Сначала создаётся строка “aba” с идентификатором 0. Далее она распечатывается, и выводится количество букв ‘a’ и ‘b’ в ней. Наконец, нулевая строка удаляется.

Затем создаются ещё две строки: строка “malloc” и строка “%d%s%” с идентификаторами 1 и 2 соответственно. Потом они распечатываются и строка “malloc” удаляется. В конце выводится количество процентов в строке “%d%s%” и она распечатывается ещё раз.

## Задача 2. Построение списка добавлением в голову

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

По заданной последовательности целых чисел построить односвязный динамический список. Каждое новое число добавлять в начало списка. Затем пройти по построенному списку и посчитать количество отрицательных чисел, входящих в список, кратных 7. После этого память освободить.

### Формат входных данных

Входной файл содержит заданную последовательность целых чисел. Числа в файле записаны через пробел. Их величина по модулю не превосходит 1000. Количество чисел может изменяться от 1 до 1000.

### Формат выходных данных

В выходной файл нужно вывести одно целое число — количество отрицательных чисел, кратных 7.

### Пример

<code>input.txt</code>	<code>output.txt</code>
10 -1 14 8 -21 -35 35 16	2

## Задача 3. Построение списка добавлением в хвост

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

По заданной последовательности целых чисел построить односвязный динамический список. Каждое новое число добавлять в конец списка. Затем пройти по построенному списку и посчитать среднее арифметическое чисел, входящих в список. После этого память освободить.

### Формат входных данных

Входной файл содержит заданную последовательность целых чисел. Числа в файле записаны через пробел. Их величина по модулю не превосходит 1000. Количество чисел может изменяться от 1 до 1000.

### Формат выходных данных

В выходной файл нужно вывести одно целое число — среднее арифметическое элементов списка.

### Пример

<code>input.txt</code>	<code>output.txt</code>
1 5 4 6 3	3

## Задача 4. Разделить на слова

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

Требуется реализовать функции для выделения слов из заданной строки. Слова отделены друг от друга символами-разделителями, которые передаются в строку как параметр.

Сигнатура функций должна быть такой:

```
typedef struct Tokens_s {
    int num;           //количество слов в строке
    char **arr;        //массив слов (каждый элемент -- строка, т.е. char*)
} Tokens;
//tokens: структура, в которую нужно положить результаты
//str: строка, в которой нужно искать слова
//delims: все символы-разделители в виде строки
void tokensSplit(Tokens *tokens, const char *str, const char *delims);
//удаляет все ресурсы внутри tokens
void tokensFree(Tokens *tokens);
```

Память для массива слов `tokens->arr` следует выделять динамически ровно на столько элементов, сколько слов в строке. Поскольку заранее это количество неизвестно, то нужно запускать алгоритм два раза: первый раз, чтобы узнать сколько слов, и второй раз, чтобы записать слова в массив.

Ваша реализация вышеописанных функций должна работать согласно следующим договорённостям:

1. Вызывающий гарантирует, что параметр `tokens` **не** нулевой (для обеих функций) и указывает на структуру `Tokens`.
2. Если при вызове `tokensSplit` указатель `tokens->arr` нулевой, то внутри функции нужно только посчитать количество слов и записать его в `tokens->num`.
3. Если при вызове `tokensSplit` указатель `tokens->arr` **не** нулевой, то он должен указывать на массив, в который точно войдут все слова. В этом случае реализация функции должна записать в `tokens->num` количество слов, а в `tokens->arr[i]` записать *i*-ое слово, самостоятельно выделив под него память с помощью `malloc`.
4. Функция `tokensFree` должна удалять массив слов и сами строки-слова с помощью `free`. При этом программа должна работать корректно, даже если эту функцию случайно вызовут два или три раза подряд.

Таким образом, вызывающий может завести структуру `tokens`, потом определить количество слов первым вызовом `tokensSplit`, затем выделить память на массив `tokens->arr`, и, наконец, найти все слова вторым вызовом `tokensSplit`.

С помощью этих функций нужно решить тестовую задачу. Дана одна строка длиной до  $10^6$ , состоящая из букв латинского алфавита и знаков препинания четырёх типов: точка, запятая, точка с запятой, двоеточие. Нужно найти слова в этой строке, состоящие из букв, и вывести их в файл в таком же формате, как показано в примере.

## Комментарий

Страница 6 из 10

## Задача 5. Построение упорядоченного списка

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

По заданной последовательности целых чисел построить односвязный динамический список. Каждое новое число добавлять в список так, чтобы он оставался упорядоченным по возрастанию. Если такое число в списке уже есть, то его не добавлять. Затем пройти по построенному списку от начала до конца и распечатать его элементы. После этого память освободить.

### Формат входных данных

Входной файл содержит заданную последовательность целых чисел. Числа в файле записаны через пробел. Их величина по модулю не превосходит 1000. Количество чисел может изменяться от 1 до 1000.

### Формат выходных данных

В выходной файл нужно вывести упорядоченную последовательность заданных чисел без повторений. Числа выводить через пробел в одну строку.

### Пример

input.txt	output.txt
10 -1 14 8 -21 -3 35 16 -3 10	-21 -3 -1 8 10 14 16 35

## Задача 6. 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 заведомая 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



( $2 \leq N, Q \leq 3 \cdot 10^5$ ). 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 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 `"%.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	001A0480
0 0.1	001A0488
0 1.1	001A0490
0 2.1	001A0498
0 3.1	001A04A0
0 4.1	00000000
0 5.1	3.1000000000000000
2 3	1.1000000000000000
2 1	001A0498
1 3	123.0000000000000000
0 123.0	00000000
2 9	
0 -1	

## 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).