NAUMEN

Тестовое задание

Стажер-разработчик

Кэширующий сервер

Владимир работает в крупной компании, одно из направлений которой обработка больших данных. Данные находятся в распределённой системе. Количество уникальных данных в системе ограничено и каждый экземпляр данных имеет свой идентификатор (номер).

Клиенты компании часто запрашивают эти данные а, время на их получение велико. Поэтому, чтобы оптимизировать обращения, Владимиру поручено написать middlware сервер (сервер посредник), через который будут проходить запросы данных.

Распределённая система хранит огромное количество данных, поэтому они не могут поместиться на сервер полностью. Однако сервер может кэшировать результаты запросов к распределённой системе: для этого выделена память, достаточная для хранения не более чем N запросов. Так как запросы от клиентов проходят через наш сервер, клиент не может получить данных напрямую из распределённой системы. Это значит, что в любом случае данные должны быть загружены на middlware сервер.

К счастью Владимира, оказалось, что самые крупные и значимые клиенты всегда обращаются за одними и теми же данными в одной и той же последовательности. Поэтому у него есть заранее определенный четкий порядок запросов. Помогите Владимиру придумать алгоритм, чтобы как можно больше данных было получено из кэша сервера, без обращения к распределённой системе.

NAUMEN

Чтобы упростить задачу считаем, что все полученные с помощью запроса данные занимают одинаковое количество байт — поэтому объём памяти для данных можно вынести из рассмотрения и оставить только количество кэшируемых запросов. Обращение к распределённой системе реализовывать не нужно, как и не нужно реализовывать оформление программы в виде сервера, нужен только алгоритм в виде консольного приложения.

В начале работы программы кэш пуст.

Требования к решению и отправке тестового задания

— Основной способ запуска программы – классический метод main;
— Метод main должен поддерживать несколько вызовов подряд;
— Алгоритм решения должен быть описан в JavaDoc основного класса (в котором находится метод main);
— Файлы с решением необходимо разместить на облачном хранилище (например Google Диск) и прикрепить ссылку к анкете на стажировку. Код не должен быть размещён в открытом источнике.

Ограничения для алгоритма (желательно)

— Ограничение по времени: 3 секунды;
— Ограничение по памяти: 64 мегабайта.

NAUMEN

Формат входных данных

— Имя входного файла: input.txt;
— Кодировка файла UTF-8;
— Файл должен находится в рабочем каталоге программы, без указания пути до файла, т.е. просто new File(“input.txt”);
— В первой строке записано 2 числа через пробел: максимальное количество запросов 1 ≤ **N** ≤ 100 000, которое может быть закэшировано на сервере, и количество запросов 1 ≤ **M** ≤ 100 000;
— Начиная со второй строки следует ровно M запросов с идентификаторами $0 \le R_i \le (2^{63}-1)$, разделённые переводом строки, т.е. M строк.

Формат выходных данных

— Имя выходного файла: output.txt;
— Кодировка файла UTF-8;
— Файл должен находится в рабочем каталоге программы, без указания пути до файла, т.е. просто new File(“output.txt”);
— В файле записано одно целое число - сколько раз пришлось обратиться к распределённой системе за данными, отсутствующими в кэше.

Оцениваться будут следующие критерии:

— Качество кода;
— Скорость алгоритма;
— Потребление памяти;
— Обработка ошибок;
— Наличие и качество тестов.

Файлы с решением размести на облачном хранилище (например Google Диск) и прикрепи ссылку к анкете на стажировку. Перед отправкой проверь доступ по ссылке и не размещай файлы в открытом доступе.
Желаем удачи!

NAUMEN

Пример

input.txt

```
5 15
3
1
4
1
5
9
2
6
5
3
5
8
7
9
3
```

output.txt

```
9
```

В первой строке файла input.txt содержится максимальное количество запросов (5), кэшируемых на сервере и количество запросов (15), разделенные пробелом.

В примере первые 3 запроса (номера 3, 1, 4) приведут к обращению к распределённой системе, так как их нет в кэше. Запрос с номером 1 есть в кэше, значит обращения к распределённой системе не будет. Запросы номеров 5 и 9 добавят их в кэш. Следующий запрос с номером 2 в кэше отсутствует, заменим номер 1 на номер 2 (так как у нас есть информация о будущих запросах, то мы видим, что запрос 1 больше не повторится и нет смысла хранить его дальше). Потом запрос номера 6 заменит неиспользуемый далее номер 2. Следующие 3 запроса будут изъяты из кэша (5, 3, 5). Далее произойдёт ещё 2 замены – 8 и 7.

Итого 9 обращений к распределённой системе.