Выполнил: Владимир Савич

Задача 1. Обоснование алгоритма

Алгоритм работы программы можно разделить на следующие части:

1. Запрос на получение тестовых данных (строки 16 – 22).
2. Открытие исходных файлов (строки 24 – 31).
3. Считывание данных из исходных файлов (строки 33 – 66).
4. Сортировка данных (строки 68 – 72, 92 – 129).
5. Запись сортированных данных в выходной файл (строки 74 – 87).

1. Запрос на получение тестовых данных.

Предполагается, что тестовые данные изначально хранятся в исходных файлах в каталоге программы в папках **“task_1_data\test_{name}\”**.

Программа выводит запрос на получение названия теста в формате заглавной буквы. К примеру, чтобы обработать программой тестовые данные из папки **“task_1_data\test_A\”**, необходимо в ответ на запрос ввести **A**.

После получения ответа на запрос программа формирует адреса исходных файлов.

2. Открытие исходных файлов.

Используя адреса файлов, сформированные в пункте 1, программа открывает исходные файлы для чтения, представляя их как объекты класса **ifstream**.

Если исходный файл невозможно открыть, программа отображает уведомление об ошибке.

3. Считывание данных из исходных файлов.

Для считывания, хранения и сортировки данных выбран контейнер вектор (**vector**).

Одним из преимуществ вектора по сравнению с массивом является отсутствие необходимости явно задавать количество элементов, под которое необходимо выделить память. Вектор способен динамически выделять память под новые элементы, фактически «удлиняя» себя во время работы программы.

Количество исходных данных в тестах различается, поэтому использование векторов для считывания данных в данном случае оправдано.

Использование массивов вместо векторов потребовало бы предварительного подсчета количества строк в исходных файлах для определения размеров массивов. Учитывая, что данные в исходных файлах рассортированы по **<id>**, определить количество строк можно следующим образом: поместить указатель чтения в конец файла, смещать его в обратном направлении до первого символа перемещения каретки, считать значение **<id>** последней строки. Однако данный метод не сработает в общем случае, когда исходные данные не сортированы по **<id>**. Использование векторов позволяет успешно считать данные и в

общем случае несортированных данных, таким образом, использование векторов в данном отношении универсально.

Программа считывает данные из исходных файлов и поочередно добавляет их в концы векторов.

Для считывания и хранения данных из файла **“players.txt”** используется вектор **vplayers_rating**. Так как данные в файле сортированы по **<user_id>**, значение **<user_id>** по сути представляется индексом элемента в векторе и остается связанным со значением **<rating>**. В общем случае, когда исходные данные не сортированы по **<user_id>**, программу несложно модифицировать, добавив еще один вектор для хранения значений **<user_id>**.

Для считывания, хранения и сортировки данных из файла **“teams.txt”** используются векторы **vteams_id** и **vteams_rating**. Вектор **vteams_id** применяется для хранения данных **<team id>**, которые затем будут сортированы относительно вектора **vteams_rating**. Вектор **vteams_rating** применяется для хранения данных о рейтингах команд. Рейтинги команд вычисляются при обращении к вектору **vplayers_rating** по значениям **<user_id>** из файла **“teams.txt”** и затем добавляются в вектор **vteams_rating**. На данном этапе значения вектора **vteams_id** соответствуют индексам элементов вектора **vteams_rating**.

После считывания данных из исходных файлов программа отбрасывает последнее значение у каждого из векторов, так как в файлах исходных данных последние строки являются пустыми. При завершении работы с файлом программа закрывает его.

4. Сортировка данных.

Для сортировки данных выбран алгоритм быстрой сортировки (**qsort**), модифицированный для векторов.

Быстрая сортировка отличается высоким быстродействием, малым потреблением памяти, простотой реализации. В полной мере алгоритм раскрывается при сортировке неупорядоченных данных. Данные в исходных файлах не упорядочены по **<rating>**, а в общем случае не упорядочены и по **<id>**. Неустойчивость сортировки в данном случае недостатком не является, так как не имеет значения, как распределятся команды с одинаковым рейтингом – главное, чтобы равномерно были сбалансированы все пары команд.

Алгоритм быстрой сортировки вызывается через рекурсивную функцию **quick_sort()**. Вектора передаются в функцию по ссылкам. Функция модифицирована так, что сортирует вектор рейтинга быстрой сортировкой (параметр **vrating**, в который передается аргумент **vteams_rating**) и одновременно относительно вектора рейтинга сортирует вектор идентификаторов (параметр **vid**, в который передается аргумент **vteams_id**).

Опорный элемент быстрой сортировки **pivot** всегда выбирается случайно на ширине отрезка, сортируемого функцией в данной итерации.

Функция разбивает вектор на отрезок с элементами, меньшими опорного, которые перемещаются до опорного элемента и отрезок с элементами, большими опорного, которые перемещаются после опорного элемента.

Затем функция рекуррентно сортирует образовавшиеся отрезки.

5. Запись сортированных данных в выходной файл.

Предполагается, что в каталоге программы создана папка **“Savich_task_1_team_pairs\”**.

Программа создает выходной файл **“Savich_task_1_team_pairs\ test_{name}_pairs.txt”**, представляя его как объект класса **ofstream**.

Программа записывает сортированные данные вектора **vteams_id** в выходной файл, выполняя перемещение на новую строку после записи каждого нечетного элемента. Для того, чтобы избежать создания пустой последней строки, последний элемент записывается в файл вне цикла.

В результате выходной файл имеет формат **“<team id> <team id>”**.

Затем программа закрывает выходной файл и завершает свою работу.