МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №4
по дисциплине «Алгоритмы и структуры данных»
ТЕМА: АЛГОРИТМЫ СОРТИРОВКИ.

Студент гр. 9381 _____ Семенов А. Н.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург
2020

Цель работы.

Ознакомление с алгоритмами сортировки линейных структур и методикой оценки эффективности алгоритмов.

Задание.

Вариант 14

Реализовать сортировку массива алгоритмом простого слияния с рекурсивной реализацией.

Основные теоретические положения.

Сортировка – последовательное расположение (упорядочивание) элементов коллекции, множества или структуры данных (линейной или нелинейной) в зависимости от выбранного критерия.

С упорядоченными элементами проще работать, чем с произвольно расположенными: легче найти необходимые элементы, исключить, вставить новые. Сортировка применяется при трансляции программ, при организации наборов данных на внешних носителях, при создании библиотек, каталогов, баз данных и т.д.

Сортировка слиянием – алгоритм сортировки, который упорядочивает списки (или другие структуры данных, доступ к элементам которых можно получать только последовательно, например, массивы) в определённом порядке. Эта сортировка — хороший пример использования принципа «разделяй и властвуй». Сначала задача разбивается на несколько подзадач меньшего размера. Затем эти задачи решаются с помощью рекурсивного вызова или непосредственно, если их размер достаточно мал. Наконец, их решения комбинируются, и получается решение исходной задачи.

Худшее, среднее и лучшее время работы алгоритма по O-символике составляет O(n*log(n)). Вспомогательные затраты памяти составляют O(n).

Алгоритм был изобретён Джоном фон Нейманом в 1945 году.

Описание алгоритма.

Решение задачи сортировки подразделяется на три этапа:

1. Сортируемый массив разбивается на две части примерно одинакового размера;
2. Каждая из получившихся частей сортируется отдельно, тем же самым алгоритмом;
3. Два упорядоченных массива половинного размера соединяются в один.

В этапах 1 и 2 рекурсивное разбиение задачи на меньшие происходит до тех пор, пока размер массива не достигнет единицы (любой массив длины 1 или 0 можно считать упорядоченным).

Описание 3 этапа – соединения двух упорядоченных массивов в один. Основную идею слияния двух отсортированных массивов можно объяснить на следующем примере. Пусть имеется два уже отсортированных по возрастанию подмассива. Тогда производится:

Слияние двух подмассивов в третий результирующий массив. На каждом шаге берется меньший из двух первых элементов подмассивов и записывается в результирующий массив. Счётчики номеров элементов результирующего массива и подмассива, из которого был взят элемент, увеличиваются на 1.

«Прицепление» остатка. Когда один из подмассивов закончился, все оставшиеся элементы второго подмассива добавляются в конец результирующего в том же порядке.

Время работы алгоритма порядка O(n*log n) при отсутствии деградации на неудачных случаях, которая является больным местом быстрой сортировки (тоже алгоритм порядка O(n*log n), но только для среднего случая). Расход памяти выше, чем для быстрой сортировки, при намного более благоприятном паттерне выделения памяти — возможно выделение одного региона памяти с самого начала и отсутствие выделения при дальнейшем исполнении. Популярная реализация требует однократно выделяемого временного буфера памяти, равного

сортируемому массиву, и не имеет рекурсий.

Достоинства и недостатки алгоритма:

Достоинства:

- ✓ Работает на структурах данных последовательного доступа;
- ✓ Хорошо сочетается с подкачкой и кэшированием памяти;
- ✓ Неплохо работает в параллельном варианте: легко разбить задачи между процессорами поровну, но трудно сделать так, чтобы другие процессоры взяли на себя работу, в случае если один процессор задержится;
- ✓ Не имеет «трудных» входных данных;
- ✓ Устойчивый - сохраняет порядок равных элементов (принадлежащих одному классу эквивалентности по сравнению).

Недостатки:

- На «почти отсортированных» массивах работает столь же долго, как на хаотичных. Существует вариант сортировки слиянием, который работает быстрее на частично отсортированных данных, но он требует дополнительной памяти, в дополнении ко временному буферу, который используется непосредственно для сортировки;
- Требует дополнительной памяти по размеру исходного массива.

Описание структур данных и функций.

1. Функции вывода информации.

`void print(T obj, ofstream* fout)` – шаблонная функция печати объекта произвольного типа *obj* в поток, адрес которого принимает в качестве аргумента *fout*, или на консоль, в случае нулевого адреса.

`void printStep(int step, ofstream* fout)` – функция, производящая печать табуляций в файл по адресу *fout* или на консоль в количестве *step*. Требуется для печати отступов, соответствующих глубине рекурсии.

`void printIntermediaArrays(int* arr_1, size_t count_1, int cur_1, int* arr_2, size_t count_2, int cur_2, int* result_arr, int`

`res_cur, int step, ofstream* fout)` – функция, производящая печать промежуточных данных, а именно отсортированных подмассивов *arr_1* и *arr_2* (количество элементов *count_1* и *count_2* соответственно) с извлеченными начальными элементами (начиная с индексов *cur_1* и *cur_2* соответственно) и результирующего массива *result_arr* с количеством *res_cur*, куда извлеченные элементы добавляются, на каждом шаге процесса слияния массивов. Печать производится в поток по адресу *fout* или на консоль, если последний – нулевой, с отступами *step*, соответствующими глубине рекурсии работы функции сортировки слиянием.

`void printResult(int* arr, int* arrFor_stdSort, size_t count, bool cmp, ofstream* p_fout)` – функция, производящая печать результата работы программы. Принимает в качестве аргументов два одинаковых, но отсортированных разными способами, массива:

arr – отсортированного слиянием,

arrFor_stdSort – отсортированного сортировкой стандартной библиотеки *std::sort()*;

количество элементов в них *count*, логическое значение *cmp*, показывающее, одинаковы ли результирующие отсортированные массивы, а также адрес потока *p_fout* для вывода.

С помощью вызова функции *print_arr()* производит печать соответствующих массивов, а также в зависимости от значения *cmp*, информацию об их равенстве.

2. Данная лабораторная работа использует структуру данных – линейный одномерный массив целых чисел (типа *int*). В данном разделе представлены функции для работы с целочисленными массивами.

`size_t input_arr(int* &arr, ifstream* fin)` – функция, производящая считывание массива с входного потока, адрес которого принимает в качестве аргумента *fin*, или с консоли, если адрес *fin* нулевой.

Для начала она считывает целое число *n* – предполагаемое количество

элементов массива. Далее в динамической памяти выделяется место под массив целых чисел в количестве *n*. В каждую ячейку массива в цикле записывается очередное целое число.

Функция кладет в переменную *arr*, переданную по ссылке адрес созданного массива, а возвращает количество его элементов.

`void print_arr(int* arr, size_t n, ofstream* fout)` – функция, производящая запись массива *arr* с *n* элементами в поток, адрес которого принимает в качестве аргумента *fout*, или на консоль, если адрес *fout* нулевой. Функция печатает массив, как ряд целых чисел, разделенных запятой с пробелом, заключенных в круглые скобки.

`int* copy_arr(int* arr, size_t n)` – функция, производящая копирование массива *arr* с *n* элементами. В функции выделяется место в динамической памяти под массив целых чисел из *n* элементов. Далее в цикле каждый элемент из исходного массива копируется в соответствующую ячейку нового. По окончании функция возвращает адрес памяти массива-копии.

`bool cmp_arr(int* arr1, int* arr2, size_t n)` – функция, производящая проверку массивов *arr_1 и arr_2* с *n* элементами на идентичность. В функции запускается цикл, в каждой итерации которого производится сравнение соответствующих элементов массивов. В случае первого же несовпадения функция возвращает *false*. В случае успешного прохода по всем элементам массивов до конца, функция возвращает *true*.

3. `void merge_sort(int* arr, size_t count, int step, ofstream* fout)` – рекурсивная функция, производящая сортировку целочисленного массива *arr* по возрастанию, с количеством элементов *count*. В качестве аргументов также принимает *step* – количество отступов для печати промежуточных данных, соответствующих глубине рекурсии, и *fout* – адрес

дескриптора потока, куда будут выводиться промежуточные данные (в случае нулевого адреса данные будут выводиться на консоль). По окончании работы функции в памяти с адресом *arr* будет находиться уже отсортированный по возрастанию массив. Функция ничего не возвращает.

Для начала в промежуточные данные производится печать информации о вызове функции для сортировки определенного массива. Вывод того или иного массива в промежуточные данные производится с помощью вызова функции *print_arr()*.

Далее отлавливается случай, когда массив состоит из одного элемента или пустой, в случае чего считается отсортированным. Тогда в промежуточные данные печатается информация об этом и о завершении функции. Функция заканчивает работу досрочно.

Затем целочисленным делением на 2 рассчитывается индекс разбиения исходного массива на две части. На его основе и учитывая адрес и длину исходного массива, рождаются адреса новых половинных подмассивов и их длины. Далее в промежуточные данные записывается информация о разбиении массива.

После чего с выводом соответствующей информации рекурсивно производится вызов функции *merge_sort()* сначала для первого, затем для второго подмассива.

Далее в динамической памяти выделяется место под вспомогательный результирующий массив длины, равной длине исходного массива. В промежуточные данные выводится информация о запуске процесса слияния двух уже отсортированных подмассивов. Образуются и обнуляются переменные индексы текущей обработки подмассивов. В промежуточные данные выводится начальное состояние подмассивов и результирующего массива, который изначально считается пустым (текущий индекс обработки результирующего массива – 0), с помощью функции *printIntermediaArrays()*.

Далее запускается цикл, в каждой итерации которого производится сравнение двух текущих элементов подмассивов (в первой итерации – начальных), и меньший из них записывается в результирующий массив, когда

индекс текущего элемента у массива, где был найден меньший элемент, и индекс результирующего массива сдвигаются на единицу вправо. Затем в промежуточные данные печатается очередное текущее состояние подмассивов (начиная с текущего индекса) и результирующего массива (до текущего индекса) с помощью функции *printIntermediaArrays()*.

Начиная с момента, когда текущий индекс одного из подмассивов совпал с его длинной, в каждой очередной итерации цикла производится добавление в результирующий массив по порядку оставшихся элементов из другого подмассива. Цикл прекращается, когда полностью заполнит результирующий массив, то есть количество итераций цикла совпадает с длиной результирующего массива.

После чего результирующий отсортированный массив выводится в промежуточные данные с информацией о присоединении остатка.

Затем запускается цикл, в каждой итерации которого элементы результирующего массива копируются в память исходного. После чего память под вспомогательный результирующий массив зачищается.

Наконец в промежуточные данные выводится информация об успешной сортировке массива и завершении функции.

4. `int main(int argc, char* argv[])` – главная функция, выполняющая программу. Она принимает массив аргументов командной строки и их количество. Если аргументов не имеется, считывание исходного массива, печать промежуточных данных и результата программы производятся на консоль. Если аргумент один, то он воспринимается как имя файла, в который будут выведены исходные данные, промежуточные данные и результат работы программы. В случае же двух аргументов, первый из них – название файла со входными данными, а второй – с выходными. В любом случае результат в краткой форме всегда выводится на консоль. Также в функции имеется проверка на корректность открытия файлов. В случае неудачного открытия файла с тем или иным названием, программа завершается с соответствующим сообщением и кодом ошибки 1.

При считывании исходных данных с файла, среди ряда целых чисел считываются только первые n, где n – стартовое число элементов массива. Если же целых чисел не хватает, то есть их меньше, чем первое число n, массив дополняется справа нулевыми элементами до длинны n. В случае отсутствия даже первого числа n в файле, оно по умолчанию будет считаться нулем.

Для начала производится проверка на корректное число аргументов: если их число превышает 3, т. е. потенциальных имен файлов больше двух, программа завершается с соответствующим сообщением и кодом ошибки 1.

Далее образуются объекты файловых потоков ввода и вывода. Создаются указатели на них соответственно, каждый из которых сперва обнуляется.

Теперь в случае наличия более одного аргумента производится попытка открытия файла для записи с названием из последнего аргумента. В случае успешного его открытия в указатель на объект потока вывода кладется его адрес.

Также проверяется наличие трех аргументов, в случае чего открывается и файл для чтения выходных данных. В случае успешного его открытия в указатель на объект потока ввода кладется его адрес.

Создается указатель на массив, который передается в функция чтения массива вместе с указателем на входной поток. Затем считанный массив выводится как исходный в поток и, в случае записи в файл, также на консоль.

Далее исходный массив копируется с помощью функции *copy_arr()*. Копия исходного массива сортируется функцией *std::sort()* из стандартной библиотеки. А затем исходный массив сортируется с помощью сортировки слиянием *merge_sort()*. После чего отсортированные разными способами массивы проверяются на идентичность с помощью функции *cmp_arr()*.

Наконец, выводится результат работы программы с помощью функции *printResult()*, которая отражает два отсортированных по-разному массива и результат их сравнения. В случае записи результата в файл, запись повторно производится на консоль вызовом той же функции. Тогда в терминал также выводится информация об имени файла, куда записались промежуточные данные.

В завершение программы память под исходный массив и ее копию

зачищаются. Файловые потоки ввода и вывода у соответствующих объектов закрываются.

Тестирование.

Основной тест – 1:

Входные данные: *4*

9 -7 4 0

Выходные данные:

Исходный массив: {9, -7, 4, 0}

Промежуточные данные:

Функция merge_sort() для массива: {9, -7, 4, 0} вызвана

Исходный массив разбивается на два подмассива: {9, -7} и {4, 0}

Сортировка первого массива:

Функция merge_sort() для массива: {9, -7} вызвана

Исходный массив разбивается на два подмассива: {9} и {-7}

Сортировка первого массива:

Функция merge_sort() для массива: {9} вызвана

В данном массиве 1 элементов, следовательно массив отсортирован

Функция merge_sort() для массива: {9} завершена

Сортировка второго массива:

Функция merge_sort() для массива: {-7} вызвана

В данном массиве 1 элементов, следовательно массив отсортирован

Функция merge_sort() для массива: {-7} завершена

Обрабатываются два отсортированных массива: {9} и {-7} таким образом, что в результирующий массив на каждом шаге записывается меньший из их первых элементов:

--

Первый массив: {9}

Второй массив: {-7}

Результирующий массив: {}

--

Первый массив: {9}

Второй массив: {}

Результирующий массив: {-7}

--

Оставшаяся часть массива присоединяется к результату:

Результирующий массив: {-7, 9}

--

Массив: {-7, 9} отсортирован. Функция merge_sort() для него завершена

Сортировка второго массива:

Функция merge_sort() для массива: {4, 0} вызвана

Исходный массив разбивается на два подмассива: {4} и {0}

Сортировка первого массива:

Функция merge_sort() для массива: {4} вызвана

В данном массиве 1 элементов, следовательно массив отсортирован

Функция merge_sort() для массива: {4} завершена

Сортировка второго массива:

Функция merge_sort() для массива: {0} вызвана

В данном массиве 1 элементов, следовательно массив отсортирован

Функция merge_sort() для массива: {0} завершена

Обрабатываются два отсортированных массива: {4} и {0} таким образом, что в результирующий массив на каждом шаге записывается меньший из их первых элементов:

--

Первый массив: {4}

Второй массив: {0}

Результирующий массив: {}

--

Первый массив: {4}

Второй массив: {}

Результирующий массив: {0}

--

Оставшаяся часть массива присоединяется к результату:

Результирующий массив: {0, 4}

--

Массив: {0, 4} отсортирован. Функция merge_sort() для него завершена

Обрабатываются два отсортированных массива: {-7, 9} и {0, 4} таким образом, что в результирующий массив на каждом шаге записывается меньший из их первых элементов:

--

Первый массив: {-7, 9}

Второй массив: {0, 4}

Результирующий массив: {}

--

Первый массив: {9}

Второй массив: {0, 4}

Результирующий массив: {-7}

--

Первый массив: {9}

Второй массив: {4}

Результирующий массив: {-7, 0}

--

Первый массив: {9}

Второй массив: {}

Результирующий массив: {-7, 0, 4}

--

Оставшаяся часть массива присоединяется к результату:

Результирующий массив: {-7, 0, 4, 9}

--

Массив: {-7, 0, 4, 9} отсортирован. Функция merge_sort() для него завершена

Результат сортировки слиянием: {-7, 0, 4, 9}

Результат стандартной сортировки std::sort(): {-7, 0, 4, 9}

Результаты стандартной сортировки std::sort() и сортировки слиянием: совпадают

Дополнительное тестирование:

Номер теста	**Входные данные**	**Выходные данные**
2	0	Результат сортировки слиянием: {} Результат стандартной сортировки std::sort(): {} Результаты стандартной сортировки std::sort() и сортировки слиянием: совпадают
3	1 -2	Результат сортировки слиянием: {-2} Результат стандартной сортировки std::sort(): {-2} Результаты стандартной сортировки std::sort() и сортировки слиянием: совпадают
4	3 3 1 9	Результат сортировки слиянием: {1, 3, 9} Результат стандартной сортировки std::sort(): {1, 3, 9}

		Результаты стандартной сортировки std::sort() и сортировки слиянием: совпадают
5	5 2 2 2 2 2	Результат сортировки слиянием: {2, 2, 2, 2, 2} Результат стандартной сортировки std::sort(): {2, 2, 2, 2, 2} Результаты стандартной сортировки std::sort() и сортировки слиянием: совпадают
6	7 -1 4 0 5 -1 -1 5	Результат сортировки слиянием: {-1, -1, -1, 0, 4, 5, 5} Результат стандартной сортировки std::sort():{-1, -1, -1, 0, 4, 5, 5} Результаты стандартной сортировки std::sort() и сортировки слиянием: совпадают
7	6 8 0 -6	Результат сортировки слиянием: {-6, 0, 0, 0, 0, 8} Результат стандартной сортировки std::sort():{-6, 0, 0, 0, 0, 8} Результаты стандартной сортировки std::sort() и сортировки слиянием: совпадают
8	3 89 25 14 90 15 100	Результат сортировки слиянием: {14, 25, 89} Результат стандартной сортировки std::sort():{14, 25, 89} Результаты стандартной сортировки std::sort() и сортировки слиянием: совпадают
9	15 10 9 8 7 6 5 4 3 2 1 0 -1 -2 -3 -4	Результат сортировки слиянием: {-4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10} Результат стандартной сортировки std::sort():{-4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10} Результаты стандартной сортировки std::sort() и сортировки слиянием: совпадают
10	8 100 200 300 400 500 550 550 1000	Результат сортировки слиянием: {100, 200, 300, 400, 500, 550, 550, 1000} Результат стандартной сортировки std::sort():{100, 200, 300, 400, 500, 550, 550, 1000}

		Результаты стандартной сортировки std::sort() и сортировки слиянием: совпадают
11	10 8 1 4 5 0 3 7 2 6 9	Результат сортировки слиянием: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} Результат стандартной сортировки std::sort():{0, 1, 2, 3, 4, 5, 6, 7, 8, 9} Результаты стандартной сортировки std::sort() и сортировки слиянием: совпадают

Вывод.

В ходе лабораторной работы было проведено ознакомление с алгоритмами сортировки линейных структур и методикой оценки эффективности алгоритмов.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл AiSD_lab_4.cpp:

```
#include <iostream>
#include <fstream>
#include <algorithm>

using namespace std;

// Функции вывода информации-----------------------------------------------------

// Функция печати объекта в поток
template <class T>
void print(T obj, ofstream* fout) {
    if (fout) *fout << obj;
    else cout << obj;
}
// Функция печати отступов
void printStep(int step, ofstream* fout){
    for (int i = 0; i < step; i++) print<string>("\t", fout);
}
```

```
// Объявление функции печати массива
void print_arr(int* arr, size_t n, ofstream* fout);
// Функция печати подмассивов слияния
void printIntermediaArrays(int* arr_1, size_t count_1, int cur_1, int* arr_2,
size_t count_2, int cur_2, int* result_arr, int res_cur, int step, ofstream*
fout){
    printStep(step, fout);
    print<string>("Первый массив: ", fout);
    print_arr(arr_1 + cur_1, count_1 - cur_1, fout);
    print<string>("\n", fout);

    printStep(step, fout);
    print<string>("Второй массив: ", fout);
    print_arr(arr_2 + cur_2, count_2 - cur_2, fout);
    print<string>("\n", fout);

    printStep(step, fout);
    print<string>("Результирующий массив: ", fout);
    print_arr(result_arr, res_cur, fout);
    print<string>("\n", fout);
    printStep(step, fout);
    print<string>("--\n", fout);
}
// Функция печати результата программы
void printResult(int* arr, int* arrFor_stdSort, size_t count, bool cmp, ofstream*
p_fout){
    print<string>("\nРезультат сортировки слиянием: ", p_fout);
    print_arr(arr, count, p_fout);
    print<string>("\n", p_fout);

    print<string>("Результат стандартной сортировки std::sort(): ", p_fout);
    print_arr(arrFor_stdSort, count, p_fout);
    print<string>("\n", p_fout);

    print<string>("Результаты стандартной сортировки std::sort() и сортировки
слиянием: ", p_fout);
    if (cmp) print<string>("совпадают\n", p_fout);
    else print<string>("не совпадают\n", p_fout);
}

// Функции ввода, вывода, копирования и сравнения массивов----------------------

// Функция считывания массива из потока
size_t input_arr(int* &arr, ifstream* fin){
```

```
    size_t n = 0;
    if (fin) *fin >> n;
    else {
        cout << "Введите число элементов массива: ";
        cin >> n;
        cout << "Введите " << n << " целых чисел по порядку через пробел (массив):
";
    }
    arr = new int[n];
    for (int i = 0; i < n; i++)
        if (fin) {
            arr[i] = 0;
            *fin >> arr[i];
        }
        else cin >> arr[i];
    return n;
}
// Определение функции печати массива в поток
void print_arr(int* arr, size_t n, ofstream* fout){
    print('{', fout);
    if (n > 0) print(arr[0], fout);
    for (int i = 1; i < n; i++){
        print<string>(", ", fout);
        print(arr[i], fout);
    }
    print('}', fout);
}
// Функция копирования массива
int* copy_arr(int* arr, size_t n){
    int* c_arr = new int[n];
    for (int i = 0; i < n; i++)
        c_arr[i] = arr[i];
    return c_arr;
}
// Функция сравнения двух массивов одинковой длины (проверки на идентичность)
bool cmp_arr(int* arr1, int* arr2, size_t n){
    for (int i = 0; i < n; i++){
        if (arr1[i] != arr2[i])
            return false;
    }
    return true;
}

// Функция сортировки слиянием-----------------------------------------------
```

```
void merge_sort(int* arr, size_t count, int step, ofstream* fout){
    printStep(step, fout);
    print<string>("Функция merge_sort() для массива: ", fout);
    print_arr(arr, count, fout);
    print<string>(" вызвана\n", fout);

    if (count == 0 || count == 1){      // Проверка на пустоту или одноэлементность массива
        printStep(step, fout);
        print<string>("В данном массиве ", fout);
        print(count, fout);
        print<string>(" элементов, следовательно массив отсортирован\n", fout);
        printStep(step, fout);
        print<string>("Функция merge_sort() для массива: ", fout);
        print_arr(arr, count, fout);
        print<string>(" завершена\n", fout);
        return;                         // Досрочное завершение функции в случае сразу отсортированного пустого или одноэлементного массива
    }
    // Рассчет параметров разбиения
    int* arr_1 = arr;                   // Адрес первого подмассива
    size_t count_1 = (int)(count / 2);  // Длина первого подмассива
    int* arr_2 = arr + count_1;         // Адрес второго подмассива
    size_t count_2 = count - count_1;   // Длинна второго подмассива

    printStep(step, fout);
    print<string>("Исходный массив разбивается на два подмассива: ", fout);
    print_arr(arr_1, count_1, fout);
    print<string>(" и ", fout);         // Печать подмассивов
    print_arr(arr_2, count_2, fout);
    print<string>("\n", fout);

    printStep(step, fout);
    print<string>("Сортировка первого массива:\n", fout);
    merge_sort(arr_1, count_1, step + 1, fout);            // Сортировка первого подмассива слиянием
    printStep(step, fout);
    print<string>("Сортировка второго массива:\n", fout);
    merge_sort(arr_2, count_2, step + 1, fout);            // Сортировка второго подмассива слиянием

    int *result_arr = new int[count];                      // Создание вспомагательного результирующего массива
```

```
    printStep(step, fout);
    print<string>("Обрабатываются два отсортированных массива: ", fout);
    print_arr(arr_1, count_1, fout);
    print<string>(" и ", fout);
    print_arr(arr_2, count_2, fout);
    print<string>(" таким образом, что в результирующий массив на каждом шаге записывается меньший из их первых элементов:\n", fout);
    // Задание начальных значений текущих индексов подмассивов слияния
    int cur_1 = 0;
    int cur_2 = 0;
    printStep(step, fout);
    print<string>("--\n", fout);
    printIntermediaArrays(arr_1,  count_1,  cur_1,  arr_2,  count_2,  cur_2, result_arr, 0, step, fout);
    for (int i = 0; i < count; i++){
        if (cur_1 < count_1 && cur_2 < count_2){
            if (arr_1[cur_1] < arr_2[cur_2]){       // Выбор меньшего элемента из текущих в подмассивах
                result_arr[i] = arr_1[cur_1];       // Занос меньшего элемента в результирующий массив
                cur_1++;                            // Сдвиг текущего индекса
            } else {
                result_arr[i] = arr_2[cur_2];       // Занос меньшего элемента в результирующий массив
                cur_2++;                            // Сдвиг текущего индекса
            }
            printIntermediaArrays(arr_1, count_1, cur_1, arr_2, count_2, cur_2, result_arr, i + 1, step, fout); // Печать состояний подмассивов
        } else {                                    // Прикрепление остатка подмассива большей длинны к результирующему массиву
            if (cur_1 == count_1){
                result_arr[i] = arr_2[cur_2];
                cur_2++;
            }
            else {
                result_arr[i] = arr_1[cur_1];
                cur_1++;
            }
        }
    }

    printStep(step, fout);
    print<string>("Оставшаяся часть массива присоединяется к результату:\n", fout);
```

```
    printStep(step, fout);
    print<string>("Результирующий массив: ", fout); // Вывод результирующего отсортированного массива
    print_arr(result_arr, count, fout);
    print<string>("\n", fout);
    printStep(step, fout);
    print<string>("--\n", fout);

    for (int i = 0; i < count; i++)                    // Перенос элементов отсортированного массива в исходный
        arr[i] = result_arr[i];
    delete[] result_arr;                               // Очистка памяти под вспомагательный массив

    printStep(step, fout);
    print<string>("Массив: ", fout);
    print_arr(arr, count, fout);
    print<string>(" отсортирован. Функция merge_sort() для него завершена\n", fout);
}
// ------------------------------------------------------------------------

// Главная функция
int main(int argc, char* argv[]){
    if (argc > 3){
        cout << "Слишком много аргументов программы\n";
        return 1;
    }

    ifstream fin;
    ifstream* p_fin = nullptr;
    ofstream fout;
    ofstream* p_fout = nullptr;
    if (argc > 1){
        if (argc == 3){
            fin.open(argv[1]);
            if (!fin){
                cout << "Ошибка открытия файла: " << argv[1] << endl;
                return 1;
            }
            p_fin = &fin;
        }

        fout.open(argv[argc - 1]);
```

```
        if (!fout){
            cout << "Ошибка открытия файла: " << argv[argc - 1] << endl;
            fin.close();
            return 1;
        }
        p_fout = &fout;
    }

    int* arr;
    size_t count = input_arr(arr, p_fin);

    print<string>("Исходный массив: ", p_fout);
    print_arr(arr, count, p_fout);
    print<string>("\n", p_fout);

    if (p_fout){
        print<string>("Исходный массив: ", nullptr);
        print_arr(arr, count, nullptr);
    }

    int* arrFor_stdSort = copy_arr(arr, count);
    std::sort(arrFor_stdSort, arrFor_stdSort + count);

    print<string>("\nПромежуточные данные:\n", p_fout);
    merge_sort(arr, count, 0, p_fout);

    bool equalityArrs = cmp_arr(arr, arrFor_stdSort, count);
    printResult(arr, arrFor_stdSort, count, equalityArrs, p_fout);
    if (p_fout) {
        printResult(arr, arrFor_stdSort, count, equalityArrs, nullptr);
        cout << "Промежуточные данные записаны в файл: " << argv[argc - 1] <<
"\n";
    }

    delete[] arr;
    delete[] arrFor_stdSort;
    fin.close();
    fout.close();
    return 0;
}
```