|  |
| --- |
| МИНОБРНАУКИ РОССИИ |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования |
| **«МИРЭА – Российский технологический университет»** |
| **РТУ МИРЭА** |
|  |

| **Отчет по выполнению практического задания № 4** | |
| --- | --- |
| **Тема:** | |
| **«Алгоритмы внешних сортировок»** | |
| Дисциплина: «Структуры и алгоритмы обработки данных» | |
|  | Выполнил студент: Жаворонкова А.А. |
|  | Группа: ИКБО-74-23 |

Москва – 2024

СОДЕРЖАНИЕ

[1 ЦЕЛЬ 3](#_gjdgxs)

[2 ЗАДАНИЕ №1 4](#_30j0zll)

[2.1 Формулировка задачи (В списке №10, Вариант 10) 4](#_1fob9te)

[2.2 Математическая модель решения алгоритма 4](#_2et92p0)

[2.2.1 Описание выполнения и блок-схема алгоритма прямого слияния 4](#_tyjcwt)

[2.2.2 Доказательство корректности циклов алгоритма прямого слияния 6](#_3dy6vkm)

[2.2.3 Определение ситуаций лучшего, среднего и худшего случая и функции роста времени работы алгоритма прямого слияния 7](#_1t3h5sf)

[2.3 Реализация алгоритма на языке C++ и проведение тестирования 8](#)

[2.3.1 Реализация алгоритма прямого слияния на языке C++ 8](#_4d34og8)

[2.3.2 Тестирование 15](#_2s8eyo1)

[2.4 Адаптация программы к новым данным 16](#_17dp8vu)

[2.6 Тестирование адаптированного кода 24](#_ziao7tuaw113)

[2.7 Практическая сложность алгоритма 27](#_1ci93xb)

[2.8 Вывод по заданию №1 27](#_dslhzqc2xurl)

[3 ЗАДАНИЕ №2 28](#_2bn6wsx)

[3.1 Формулировка задачи 28](#_qsh70q)

[3.2 Математическая модель решения алгоритма 28](#_u290pxmzmczz)

[3.2.1 Описание выполнения и блок-схема алгоритма естественного слияния 28](#_ujf0wkarvz46)

[3.2.2 Доказательство корректности циклов алгоритма естественного слияния 30](#_vg00rxuqjnjx)

[3.2.3 Определение ситуаций лучшего, среднего и худшего случая и функции роста времени работы алгоритма естественного слияния 31](#_foeym3wsnfua)

[3.3 Реализация алгоритма на языке C++ и проведение тестирования 32](#_44wcxh77t5yt)

[3.3.1 Реализация алгоритма естественного слияния на языке C++ 32](#_fweoj2zgp349)

[2.3.2 Тестирование 39](#_qzyyeyy25qpm)

[3.4 Адаптация программы к новым данным 40](#_tztmvu7f1qy2)

[3.6 Тестирование адаптированного кода 50](#_omt9yh4vlatz)

[3.7 Практическая сложность алгоритма 61](#_80lopqn0frwv)

[3.8 Вывод по заданию №2 61](#_fc3bwntftrcf)

[5 ВЫВОДЫ 63](#_ihv636)

[6 ЛИТЕРАТУРА 64](#_32hioqz)

# **1 ЦЕЛЬ**

Освоить приёмы сортировки данных из файлов

# **2 ЗАДАНИЕ №1**

## **2.1 Формулировка задачи (В списке №10, Вариант 10)**

Разработать программу и применить алгоритм внешней сортировки прямого слияния к сортировке файла данных “Магазин игрушек” по значению ключевого поля “название”.

1) Реализовать функцию сортировки (возможно, с вспомогательными функциями) и основную подпрограмму main.

2) Отладить программу, протестировать на примере из п.2.

3) Предварительно подготовить файл данных в соответствии с вариантом (не менее 32 записей).

4) Адаптировать программу для сортировки файла с записями, протестировать на подготовленном ранее файле.

5) Определить практическую сложность алгоритма для файлов с увеличивающимся количеством записей (8, 16, 32). Сформировать таблицу результатов, указав количество записей и время сортировки.

## **2.2 Математическая модель решения алгоритма**

### **2.2.1 Описание выполнения и блок-схема алгоритма прямого слияния**

Внешняя сортировка — сортировка больших объёмов данных во внешней памяти (например, в файлах), не вмещающихся в оперативную память, то есть когда применить одну из внутренних сортировок невозможно. Стоит отметить, что внутренняя сортировка значительно эффективней внешней, так как на обращение к оперативной памяти затрачивается намного меньше времени, чем к магнитным дискам, лентам и т. п.

Основной принцип сортировки слиянием такой: делим массив пополам, каждый из них сортируем слиянием и потом соединяем оба массива. Каждый разделённый массив тоже нарезаем на два подмассива до тех пор, пока в каждом не окажется по одному элементу.

Алгоритм сортировки прямым слиянием является простейшим алгоритмом внешней сортировки, основанный на процедуре слияния серией. В данном алгоритме длина серий фиксируется на каждом шаге. В исходном файле все серии имеют длину 1, после первого шага она равна 2, после второго – 4, после третьего – 8, после k -го шага – 2k.

Алгоритм сортировки простым слиянием:

1. Исходный файл f разбивается на два вспомогательных файла f1 и f2.

2. Вспомогательные файлы f1 и f2 сливаются в файл f, при этом одиночные элементы образуют упорядоченные пары.

3. Полученный файл f вновь обрабатывается, как указано в шагах 1 и 2. При этом упорядоченные пары переходят в упорядоченные четверки.

4. Повторяя шаги, сливаем четверки в восьмерки и т.д., каждый раз удваивая длину слитых последовательностей до тех пор, пока не будет упорядочен целиком весь файл.

После выполнения i проходов получаем два файла, состоящих из серий длины 2i. Окончание процесса происходит при выполнении условия 2i>=n. Следовательно, процесс сортировки простым слиянием требует порядка O (log n) проходов по данным.

Признаками конца сортировки простым слиянием являются следующие условия:

1. длина серии не меньше количества элементов в файле (определяется после фазы слияния);

2. количество серий равно 1 (определяется на фазе слияния).

3. количество серий равно 1 (определяется на фазе слияния).

Реализация данного описания выполнения алгоритма представлена в виде блок-схемы (рис.1).

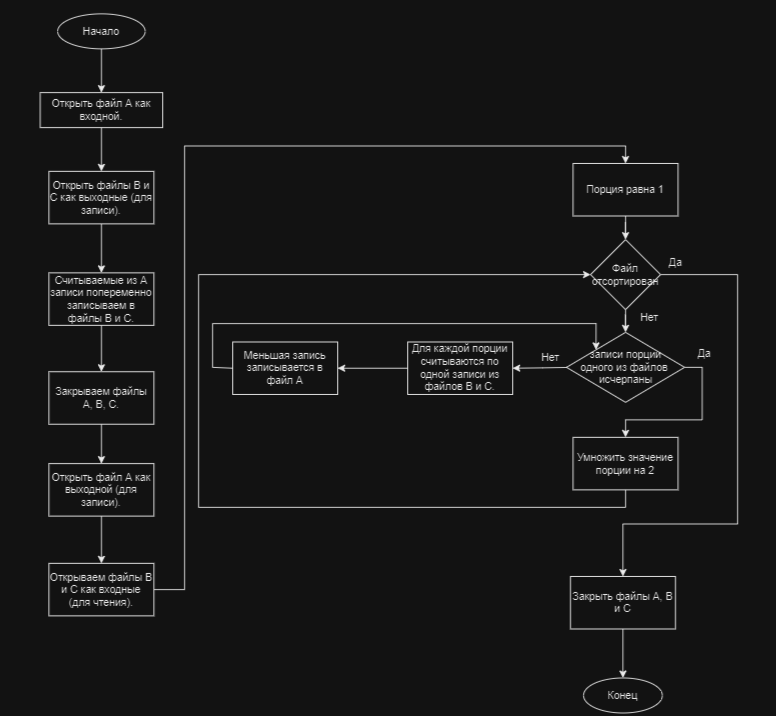


Рисунок 1 – Блок-схема алгоритма прямого слияния

### **2.2.2 Доказательство корректности циклов алгоритма прямого слияния**

Инвариант для цикла: На каждом шаге сортировки, при склейке двух упорядоченных подмассивов, получается упорядоченный подмассив большего размера.

Докажем конечность цикла. Алгоритм начинается с разбиения исходного списка на подсписки размером 1 элемент. Затем происходит слияние этих подсписков по парам. По мере продвижения алгоритма, подсписки сортируются и сливаются в все большие подсписки. Процесс повторяется до тех пор, пока весь список не будет отсортирован.

На каждом шаге алгоритма размер подсписков удваивается, пока весь список не будет полностью упорядочен. Таким образом, количество шагов, необходимых для завершения алгоритма, ограничено логарифмом размера исходного списка.

Из этого следует, что циклы прямого слияния имеют конечную сложность и всегда завершаются в конечное время, даже для очень больших списков.

Из доказательства можно сделать вывод, что все циклы данного алгоритма корректны.

### **2.2.3 Определение ситуаций лучшего, среднего и худшего случая и функции роста времени работы алгоритма прямого слияния**

В лучшем случае прямое слияние будет иметь время выполнения O(n log n), где n - количество элементов в исходном массиве. Лучший случай возникает, когда массив уже отсортирован и не требует дополнительных операций по перемещению и слиянию элементов. В этом случае, алгоритм просто делит массив на две части и объединяет их без необходимости в дополнительных операциях.

В худшем случае прямое слияние также имеет время выполнения O(n log n). Худший случай возникает, когда массив полностью неотсортирован или отсортирован в обратном порядке. В этом случае, алгоритм будет снова разделять исходный массив на две части и рекурсивно сортировать их, а затем сливать две половины вместе. Хотя это требует больше операций по слиянию, время выполнения остается O(n log n).

В среднем случае время выполнения прямого слияния также является O(n log n). В среднем случае, массив может быть любым, и алгоритм все равно будет обеспечивать эффективность O(n log n).

Для данного метода сортировки, время исполнения в худшем, среднем и лучшем случае увеличивается квазилинейно с ростом размера входного массива. вне зависимости от входных данных, время выполнения алгоритма прямого слияния будет составлять O(n log n), что делает его одним из эффективных алгоритмов сортировки для больших объемов данных.

Ёмкостная сложность алгоритма будет равна O(n).

## **2.3 Реализация алгоритма на языке C++ и проведение тестирования**

### **2.3.1 Реализация алгоритма прямого слияния на языке C++**

Реализуем данный алгоритм на языке C++(блок кода 1). Для реализации понадобятся такие библиотеки, как iostream, random, chrono, fstream, algorithm, vector, string.

iostream: библиотека в C++, являющаяся заголовочным файлом с классами, функциями и переменными для предоставления ввода и вывода данных через стандартные потоки cout (для вывода на консоль) и cin (для ввода с консоли).

chrono: библиотека в C++, используемая для работы с временем и временными измерениями, включая функции для измерения времени исполнения кода.

fstream: библиотека в C++, предоставляющая возможности для работы с файлами через потоки (ввода и вывода), позволяет записывать данные в файлы и читать данные из файлов.

algorithm: библиотека в C++, содержащая широкий набор стандартных алгоритмов, таких как сортировка, поиск, преобразование контейнеров и другие.

vector: библиотека в C++, представляющая собой динамический массив, позволяющий хранить и управлять элементами одного типа данных.

string: библиотека в C++, предоставляющая функции для работы с символьными строками, включая конкатенацию, поиск, сравнение и другие операции.

pragma once: это нестандартная, но широко поддерживаемая директива препроцессора, предназначенная для включения текущего заголовочного файла только один раз в одну компиляцию. Таким образом, #pragma когда-то служила той же цели, что и include guards, но с рядом преимуществ, включая меньшее количество кода, предотвращение конфликтов имен и иногда повышение скорости компиляции.

В программе также будут использоваться классы (в C++ — это абстракция, описывающая методы и свойства ещё не существующих объектов) и объекты (это конкретное представление абстракции, имеющее свои свойства и методы. Созданные объекты на основе одного класса называются экземплярами этого класса.)

| #include <iostream> #include <fstream> #include <algorithm> #include <vector> #include <string> using namespace std;   *//Оператор перегрузки вывода для потока ostream и вектора целых чисел* ostream& operator<< (ostream& lhv, const vector<int>& rhv) {  const auto size = rhv.size();*// Получение размера вектора*  lhv << "(";  for (int i = 0; i < size; ++i)*// Цикл вывода элементов вектора с пробелами между ними*  {  lhv << rhv[i];  if(i+1 < size) lhv << " ";  }  lhv << ")";  return lhv;*// Возврат потока вывода* } class SExternalMergeSort// Объявление класса SExternalMergeSort { public:  *// way\_count -- количество путей (вспомогательных файлов) сортировки*  explicit SExternalMergeSort(const string& filename, unsigned int way\_count = 2) : seriesLength\_{ 0 }, wayCount\_ { way\_count }, inputFilename\_{ filename } {  if (wayCount\_ < 2) wayCount\_ = 2;*// Проверка и коррекция значения wayCount, если оно меньше 2*  };  virtual ~SExternalMergeSort() = default;*// Закрытие виртуального деструктора по умолчанию*  void sort(fstream& input);*// Определение функции sort класса SExternalMergeSort* private:  *// вычисление длины серии*   void computeSeriesLength(fstream& input);  *// получить список из wayCount\_ вспомогательных файлов*  vector<fstream> getExternalFiles();  *// закрыть и открыть вспомогательные файлы для корректного ввода-вывода*  void reopenExternalFiles(vector<fstream>& ext, ios\_base::openmode mode);  *// удалить вспомогательные файлы*  void removeExternalFiles();  *// закрыть и открыть файл входных данных для корректного ввода-вывода*  void reopenInputFile(fstream& input, ios\_base::openmode mode);  *// разделение серии во вспомогательные файлы подгруппами длинны subgroup\_length*  void split(fstream& input, vector<fstream>& ext, const unsigned int subgroup\_length);  *// слияние из вспомогательных файлов подгруппами длинны subgroup\_length*  void merge(fstream& input, vector<fstream>& ext, const unsigned int subgroup\_length);   unsigned int seriesLength\_; *// длина серии вычисляется при запуске сортировки*   unsigned int wayCount\_; *// количество путей (вспомогательных файлов) сортировки*   string inputFilename\_; }; int main(int argc, char\*\* argv) {  const auto input\_file = "A";*// Определение строки inputfile с именем "A"*  fstream input(input\_file, ios\_base::in | ios\_base::out);*//Открытие файла input в режиме чтения и записи*  if (!input) *//Проверка, удалось ли открыть файл*  {  perror("Error with input file");  return EXIT\_FAILURE;  }  SExternalMergeSort(input\_file).sort(input);*//Создание объекта SExternalMergeSort и вызов метода sort с передачей input в качестве параметра*  input.close();*//Закрытие файла input*  system("pause");*//Выполнение команды system("pause")*  return EXIT\_SUCCESS;*//Возврат значения EXITSUCCESS* } void SExternalMergeSort::computeSeriesLength(fstream& input\_series) {  this->seriesLength\_ = 0;*//Обнуление переменной seriesLength*  int value = 0;*//Объявление переменной value для хранения числа из файла*  while (input\_series >> value)*//Цикл чтения чисел из файла и увеличение значения seriesLength*  {  this->seriesLength\_++;  } } *//Определение функции getExternalFiles класса SExternalMergeSort* vector<fstream> SExternalMergeSort::getExternalFiles() {  vector<fstream> ext(this->wayCount\_);*//Создание вектора ext с размером wayCount*  for(unsigned int i = 0; i < this->wayCount\_; ++i)*//Цикл создания вспомогательных файлов и добавления их в вектор*  {  const auto filename = to\_string(i + 1);  ofstream f(filename);  ext[i] = fstream(filename);  }  return ext; } *//Определение функции reopenExternalFiles класса SExternalMergeSort* void SExternalMergeSort::reopenExternalFiles(vector<fstream>& ext, ios\_base::openmode mode) {  for (int i = 0; i < ext.size(); ++i)*//Цикл закрытия и переоткрытия вспомогательных файлов с учетом режима mode*  {  ext[i].close();   const auto filename = to\_string(i + 1);  ext[i].open(filename, mode);  } } *// Определение функции removeExternalFiles класса SExternalMergeSort* void SExternalMergeSort::removeExternalFiles() {  for (unsigned int i = 0; i < this->wayCount\_; ++i)*//Цикл удаления вспомогательных файлов с выводом результата*  {  const auto filename = to\_string(i + 1);  const auto removeResult = remove(filename.c\_str());  cout << "Remove external file (" << filename << "): " << boolalpha << removeResult << endl;  } } *// закрыть и открыть файл входных данных для корректного ввода-вывода* void SExternalMergeSort::reopenInputFile(fstream& input, ios\_base::openmode mode) {  input.close();  input.open(this->inputFilename\_, mode); } *// Определение функции split класса SExternalMergeSort* void SExternalMergeSort::split(fstream& input\_series, vector<fstream>& ext, const unsigned int subgroup\_length) {  while(!input\_series.eof())*//Цикл чтения данных из входного файла и записи в вспомогательные файлы по подгруппам*  {  for(auto&& f: ext)*//Цикл для сброса буфера каждого вспомогательного файла после записи данных*  {  int value = 0;  unsigned int step = 0;  while(input\_series >> value)  {  f << value << " ";  cout << value << "\t";  ++step;  if (step == subgroup\_length) break;  }  cout << " | ";  }  cout << endl;  }  for (auto&& f : ext)  {  f.flush();  } } *//Определение функции merge класса SExternalMergeSort, принимающей файловый поток inputseries, вектор файловых потоков ext и длину подгруппы subgrouplength* void SExternalMergeSort::merge(fstream& input\_series, vector<fstream>& ext, const unsigned int subgroup\_length) {  if (subgroup\_length == 0) return;*//Проверка, если длина подгруппы равна 0, то выход из функции*  vector<int> buf;*//Создание вектора buf для временного хранения данных с резервированием необходимого размера*  buf.reserve(subgroup\_length \* this->wayCount\_);*//Вычисление количества полных групп в путях*  *// количество полных групп в путях*  const unsigned int fullGroupCount = this->seriesLength\_ / subgroup\_length / this->wayCount\_;  *// учет возможного отсутствия последней группы в некоторых путях*  const unsigned int add = this->seriesLength\_ % this->wayCount\_ == 0 ? 0 : 1;  *// количество шагов алгоритма*  const unsigned int stepsCount = fullGroupCount + add;  unsigned int step = 0;  while(step < stepsCount)*//Цикл по шагам выполнения алгоритма*  {  for (auto& f: ext)*//Цикл по каждому файловому потоку вектора ext*  {  int value = 0;  unsigned int substep = 0;  while(substep < subgroup\_length && f >> value)  {  buf.push\_back(value);  ++substep;  }  }  std::sort(begin(buf), end(buf));*//Сортировка данных в буфере buf*  const auto bufLength = buf.size();*//Получение длины буфера buf*  for(int i = 0; i < bufLength; ++i)*//Цикл вывода отсортированных данных из буфера в файл inputseries и на консоль*  {  input\_series << buf[i] << " ";  cout << buf[i] << " ";  }  cout << "\t";  buf.clear();*//Очистка буфера buf и увеличение значения шага на единицу*  buf.resize(0);*//Вывод символа новой строки на консоль*  step++;  }  cout << endl; } *//Определение функции sort класса SExternalMergeSort* void SExternalMergeSort::sort(fstream& input\_series) {   this->computeSeriesLength(input\_series);*//Вычисление длины серии данных inputseries*  if (this->seriesLength\_ <= 1)*//Проверка, если длина серии меньше или равна 1, то выход из функции*  {  return;  }  vector<fstream> ext = this->getExternalFiles();*//Получение вектора вспомогательных файлов ext*  *//Цикл сортировки по подгруппам, удваивая длину подгруппы на каждой итерации*  for(unsigned int subgroup\_length = 1; subgroup\_length < this->seriesLength\_; subgroup\_length \*= 2)  {  reopenInputFile(input\_series, ios\_base::in);  reopenExternalFiles(ext, ios\_base::out);  cout << "Subgroup with length " << to\_string(subgroup\_length) << ":" << endl;*// Переоткрытие входного файла и вспомогательных файлов для чтения и записи*  this->split(input\_series, ext, subgroup\_length);*//Вывод информации о текущей длине подгруппы на консоль*  reopenInputFile(input\_series, ios\_base::out);*// Разделение данных на подгруппы и запись в вспомогательные файлы*  reopenExternalFiles(ext, ios\_base::in);*//Переоткрытие входного файла и вспомогательных файлов для обратного чтения и записи*  cout << "Merge to groups with length " << to\_string(subgroup\_length) << endl;*//Вывод информации о слиянии групп с текущей длиной подгруппы*  this->merge(input\_series, ext, subgroup\_length);*// Вызов функции merge для слияния данных из вспомогательных файлов входного файла*  }  input\_series.flush();*//Сброс буфера входного файла*  this->removeExternalFiles();*//Удаление вспомогательных файлов после завершения сортировки.* } |
| --- |

Блок кода 1 – Программа main.cpp

### **2.3.2 Тестирование**

Стоит задача протестировать программу с заданным данными в файле А: 8 2 13 4 15 6 9 11 3 7 5 10 1 12 14. Для реализации алгоритма будем использовать два файла, в которые будем разливать данные файла: файл В и файл С. Сначала разбиваем по одному элементу: B: 8 13 15 9 3 5 1 14 и C: 2 4 6 11 7 10 12. Сливаем в упорядоченные двойки: А: 2 8 4 13 6 15 9 11 3 7 5 10 1 12 14 2. Разливаем по два: B: 2 8 6 15 3 7 1 12 и C: 4 13 9 11 5 10 14 Сливаем в упорядоченные четверки и т. д. пока длина порции не станет равной длине массива. Воспользуемся структурой high\_resolution\_clock для подсчёта затраченного времени на сортировку. Для более точных результатов в программе будем рассматривать микросекунды. Результаты тестирования представлены на рисунке 2.

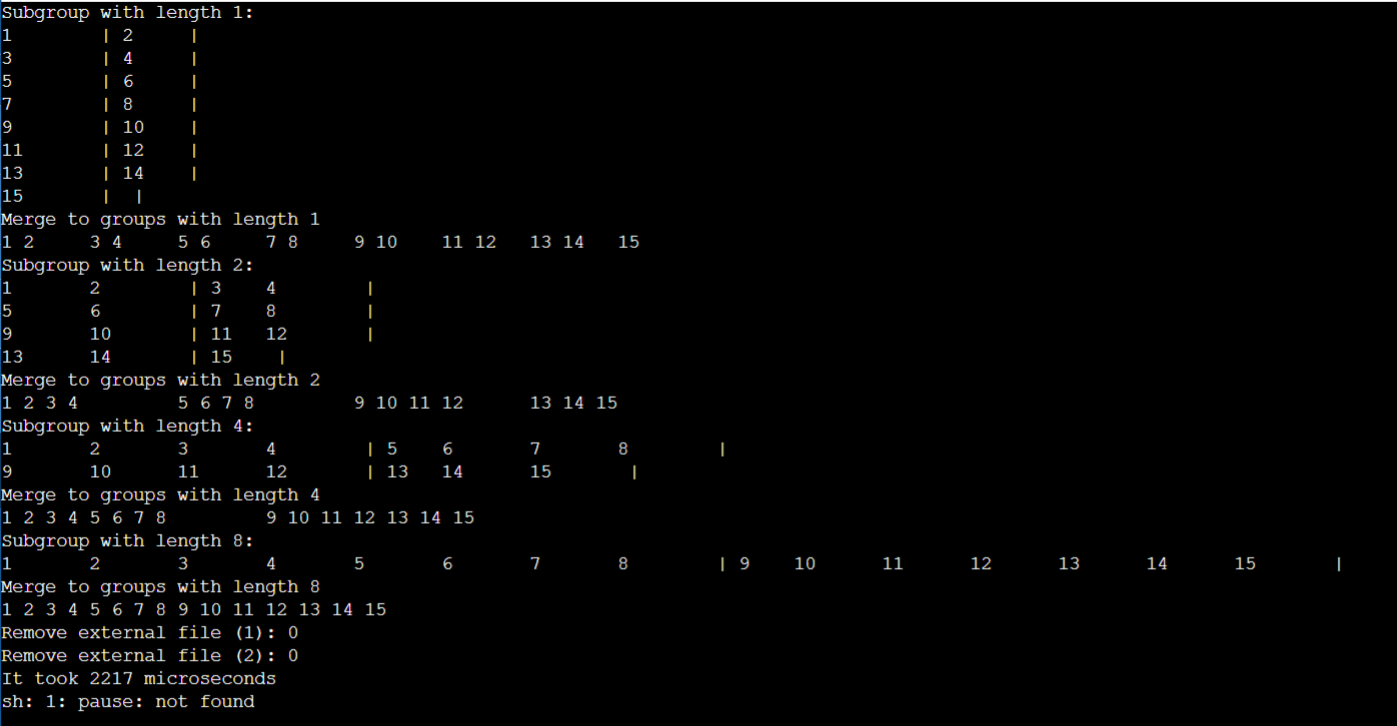


Рисунок 2 - Тестирование программы с входными данными из примера

## **2.4 Адаптация программы к новым данным**

Адаптируем программу к данным по условию индивидуального варианта. Магазин игрушек. Сведения об игрушке: Название (например: кукла, конструктор и т.д.), стоимость, возрастные границы детей (для кого игрушка предназначена) два поля – начальный возраст и конечный). То, что подчёркнуто, является ключом для сортировки. Зададим структуру файла, которая должна состоять из string (последовательности символов, хранящихся в массиве char) и трёх целочисленных значений.

Будет использоваться пользовательский заголовочный файл "ems\_simple.h".

Реализация кода состоит из двух частей. Из заголовочного файла "ems\_simple.h"(блок кода 2) и исходного файла main.cpp(блок кода 3).

| #pragma once #include <iostream> #include <fstream> #include <algorithm> #include <vector> #include <string> #include <functional> using namespace std; *// SType - тип сортируемых данных* *// DSep - разделитель данных в файле* template<class SType, char DSep = ' '> class SExternalMergeSort { public:  *// way\_count -- количество путей (вспомогательных файлов) сортировки*  explicit SExternalMergeSort(const std::string& filename, unsigned int way\_count = 2) : seriesLength\_{ 0 }, wayCount\_{ way\_count }, inputFilename\_{ filename } {  if (wayCount\_ < 2) wayCount\_ = 2;  };  virtual ~SExternalMergeSort() = default;  template<class Compare>  void sort(fstream& input, Compare&& cmp = [](const SType& a, const SType& b) { return a < b; });  private:  *// вычисление длины серии*   void computeSeriesLength(std::fstream& input);  *// имя для вспомогательного файла*  string getExternalFilename(const unsigned int way);  *// получить список из wayCount\_ вспомогательных файлов*  vector<fstream> getExternalFiles();  *// закрыть и открыть вспомогательные файлы для корректного ввода-вывода*  void reopenExternalFiles(vector<fstream>& ext, ios\_base::openmode mode);  *// удалить вспомогательные файлы*  void removeExternalFiles();  *// закрыть и открыть файл входных данных для корректного ввода-вывода*  void reopenInputFile(fstream& input, ios\_base::openmode mode);  *// разделение серии во вспомогательные файлы подгруппами длинны subgroup\_length*  void split(fstream& input, vector<fstream>& ext, const unsigned int subgroup\_length);  *// слияние из вспомогательных файлов подгруппами длинны subgroup\_length*  template<class Compare>  void merge(fstream& input, vector<fstream>& ext, const unsigned int subgroup\_length, Compare& cmp);  unsigned int seriesLength\_; *// длина серии вычисляется при запуске сортировки*   unsigned int wayCount\_;*// количество путей (вспомогательных файлов) сортировки*   std::string inputFilename\_; }; template<class SType, char DSep> // Вычисляет длину серии данных, считывая их из файла input\_series до конца. Кол-во прочитанных элементов сохраняется в переменной seriesLength\_. void SExternalMergeSort<SType, DSep>::computeSeriesLength(fstream& input\_series) {  this->seriesLength\_ = 0;  SType value{};  while (input\_series >> value)  {  this->seriesLength\_++;  } }  template<class SType, char DSep> // Возвращает строку, представляющую внешний файл с данными, в зависимости от переданного параметра way. Файлы нумеруются с единицы. string SExternalMergeSort<SType, DSep>::getExternalFilename(const unsigned int way) {  return "/" + to\_string(way + 1); }  template<class SType, char DSep> // Создает и открывает вектор внешних файлов нужного размера и возвращает его. Названия файлов формируются с помощью функции getExternalFilename. vector<fstream> SExternalMergeSort<SType, DSep>::getExternalFiles() {  vector<fstream> ext(this->wayCount\_);  for (unsigned int i = 0; i < this->wayCount\_; ++i)  {  const auto filename = this->getExternalFilename(i);  ofstream f(filename);  ext[i] = fstream(filename);  }  return ext; }  template<class SType, char DSep> //Закрывает и повторно открывает все внешние файлы из вектора ext с заданным режимом mode. Названия файлов берутся из функции getExternalFilename. void SExternalMergeSort<SType, DSep>::reopenExternalFiles(vector<fstream>& ext, ios\_base::openmode mode) {  for (int i = 0; i < ext.size(); ++i)  {  ext[i].close();   const auto filename = getExternalFilename(i);  ext[i].open(filename, mode);  } }  template<class SType, char DSep> //Удаляет внешние файлы, используемые для сортировки. void SExternalMergeSort<SType, DSep>::removeExternalFiles() {  for (unsigned int i = 0; i < this->wayCount\_; ++i)  {  const auto filename = getExternalFilename(i);  const auto removeResult = remove(filename.c\_str());  cout << "Remove external file (" << filename << "): " << boolalpha << removeResult << endl;  } }  template<class SType, char DSep> //Повторно открывает входной файл с указанным режимом. void SExternalMergeSort<SType, DSep>::reopenInputFile(fstream& input, ios\_base::openmode mode) {  input.close();  input.open(this->inputFilename\_, mode); }  template<class SType, char DSep> //Разделяет данные из входного файла на подгруппы и записывает их во внешние файлы. void SExternalMergeSort<SType, DSep>::split(fstream& input\_series, vector<fstream>& ext, const unsigned int subgroup\_length) {  while (!input\_series.eof())  {  for (auto&& f : ext)  {  SType value{};  unsigned int step = 0;  while (input\_series >> value)  {  f << value << DSep;  cout << value << "\t";  ++step;  if (step == subgroup\_length) break;  }  cout << " | ";  }  cout << endl;  } } template<class SType, char DSep> //Выполняет слияние подгрупп и сортировку данных по возрастанию. template<class Compare> void SExternalMergeSort<SType, DSep>::merge(  fstream& input\_series, vector<fstream>& ext,  const unsigned int subgroup\_length,  Compare& cmp) {  if (subgroup\_length == 0) return;  vector<SType> buf;  buf.reserve(subgroup\_length \* this->wayCount\_);  *// количество полных групп в путях*  const unsigned int fullGroupCount = this->seriesLength\_ / subgroup\_length / this->wayCount\_;  *// учет возможного отсутствия полной последней группы в некоторых путях*  const unsigned int add = (this->seriesLength\_ / subgroup\_length) % this->wayCount\_ == 0 ? 0 : 1;  *// количество шагов алгоритма*  const unsigned int stepsCount = fullGroupCount + add;  unsigned int step = 0;  while (step < stepsCount)  {  for (auto& f : ext)  {  SType value{};  unsigned int substep = 0;  while (substep < subgroup\_length && f >> value)  {  buf.push\_back(value);  ++substep;  }  }  std::sort(begin(buf), end(buf), cmp);  const auto bufLength = buf.size();  for (int i = 0; i < bufLength; ++i)  {  input\_series << buf[i] << DSep;  cout << buf[i] << DSep;  }  cout << "\t";  *//Очищаем буфер и увеличиваем счетчик шага.*  buf.clear();  buf.resize(0);  step++;  }  cout << endl; } template<class SType, char DSep> //Сортирует данные во внешних файлах с использованием алгоритма внешней сортировки template<class Compare> void SExternalMergeSort<SType, DSep>::sort(fstream& input\_series, Compare&& cmp) {  this->computeSeriesLength(input\_series);  if (this->seriesLength\_ <= 1)*//Если длина меньше или равна 1, то возвращаемся.*  {  return;  }   auto&& cmp\_ = forward<Compare>(cmp);  vector<fstream> ext = this->getExternalFiles();  *// Сортируем пока длина подгруппы меньше длины серии*  for (unsigned int subgroup\_length = 1; subgroup\_length < this->seriesLength\_; subgroup\_length \*= 2)  {  reopenInputFile(input\_series, ios\_base::in);  reopenExternalFiles(ext, ios\_base::out);  cout << "Subgroup with length " << to\_string(subgroup\_length) << ":" << endl;  this->split(input\_series, ext, subgroup\_length);  reopenInputFile(input\_series, ios\_base::out);  reopenExternalFiles(ext, ios\_base::in);  cout << "Merge to groups with length " << to\_string(subgroup\_length) << endl;  this->merge(input\_series, ext, subgroup\_length, cmp\_);  }  *//Сбрасываем буфер в файл и удаляем внешние файлы.*  input\_series.flush();  this->removeExternalFiles(); } |
| --- |

Блок кода 2 - заголовочный файл "ems\_simple.h"

| #include <fstream> #include <vector>  #include <chrono> *//Подключение пользовательского заголовочного файла "ems\_simple.h"* #include "ems\_simple.h" using namespace std; *//Определение структуры файла A, содержащего поля n, a, b, c.* struct A {  string n;*//Название игрушки*  int a = 0;*//цена игрушки*  int b = 0;*//возраст начальный*  int c = 0;*//возраст конечный* }; *//Перегрузка оператора >> для чтения значений типа* istream& operator>>(istream& lhv, A& rhv) {  string n;  while (true)  {  const auto c = lhv.get();  if (c == -1 || c == ',') break;  if (c >= 0) n += (char)c;  }  rhv.n = n;*// Присваиваем свойству "n" объекта "rhv" значение переменной "n" rhv.n = n*  int value = 0;*// Создаем переменную "value" и считываем значение из потока "lhv" int value = 0; lhv >> value;*  lhv >> value;*// Считываем значение а переменную "value" из потока "lhv" lhv >> value*  rhv.a = value;*// Присваиваем свойству "a" объекта "rhv" значение переменной "value", считанной из потока "lhv" rhv.a = value*  lhv.get();*// Пропускаем символ '\n' в потоке "lhv" lhv.get()*  lhv >> value;*// Снова считываем значение в переменную "value" из потока "lhv" lhv >> value*  rhv.b = value;*// Присваиваем свойству "b" объекта "rhv" значение переменной "value" rhv.b = value*  lhv.get();*// Пропускаем символ '\n' в потоке "lhv" lhv.get()*  lhv >> value;*// Снова считываем значение в переменную "value" из потока "lhv" lhv >> value*  rhv.c = value;*// Присваиваем свойству "c" объекта "rhv" значение переменной "value" rhv.c = value*  lhv.get(); *// считывание*  return lhv; } *//Перегрузка оператора << для записи значений типа A* ostream& operator<<(ostream& lhv, const A& rhv) {  lhv << rhv.n << ", " << rhv.a << ", " << rhv.b << ", " << rhv.c;  return lhv; } int main(int argc, char\*\* argv) {  setlocale(LC\_ALL, "RUS");  auto start = chrono::high\_resolution\_clock::now();*// Начало сортировки*  *//Открытие файла "A" в режиме чтения и записи.*  const auto input\_file = "A";  fstream input(input\_file, ios\_base::in | ios\_base::out);  *//Проверка успешности открытия файла*  if (!input)  {  perror("Error with input file");  return EXIT\_FAILURE;  }  *//Использование шаблонной функции NExternalMergeSort из заголовочного файла "ems\_simple.h"*  SExternalMergeSort<A, '\n'>(input\_file).sort(input, [](const A& a, const A& b) { return a.n < b.n; });  input.close();*//Закрытие входного файла*  auto end = chrono::high\_resolution\_clock::now();*// Конец сортировки*  auto d = chrono::duration\_cast<chrono::microseconds>(end - start).count();*// затраченное время*  cout << "It took " << d << " microseconds" << endl;  system("pause");*//Приостановка выполнения программы для ожидания ввода пользователя*   return EXIT\_SUCCESS;*//Возврат успешного завершения программы.* } |
| --- |

Блок кода 3 - Файл main.cpp

## **2.6 Тестирование адаптированного кода**

Стоит задача протестировать программу с заданным данными в файле А. Воспользуемся структурой high\_resolution\_clock для подсчёта затраченного времени на сортировку. Для более точных результатов в программе будем рассматривать микросекунды. Результаты тестирования будут продемонстрированы на рисунке 3-7.

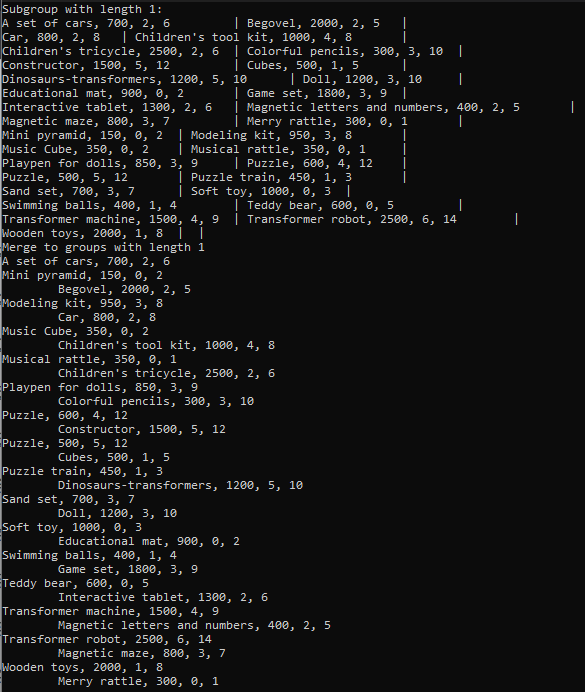


Рисунок 3 - Тестирование адаптированной программы

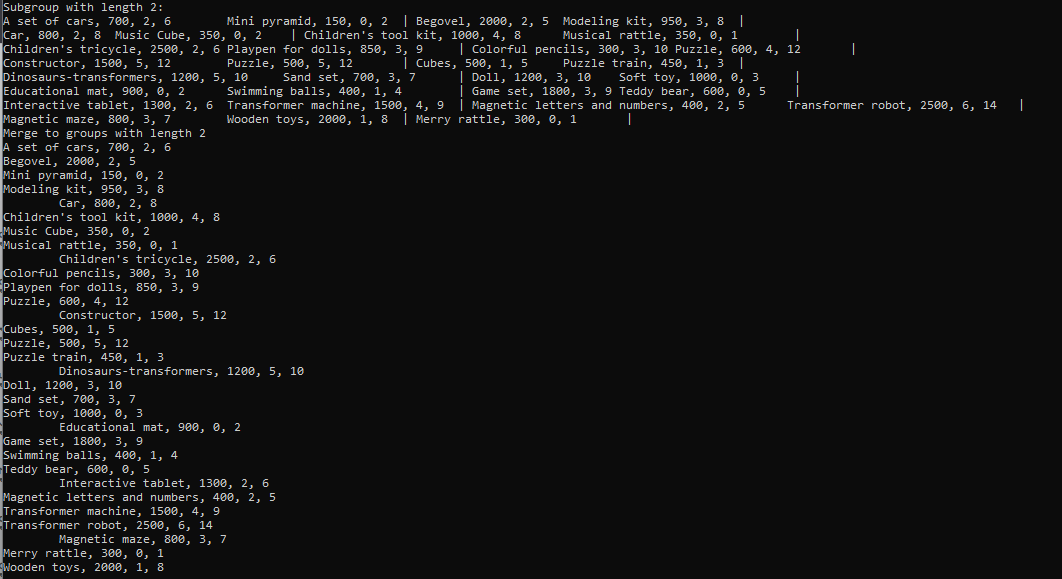


Рисунок 4 - Тестирование адаптированной программы

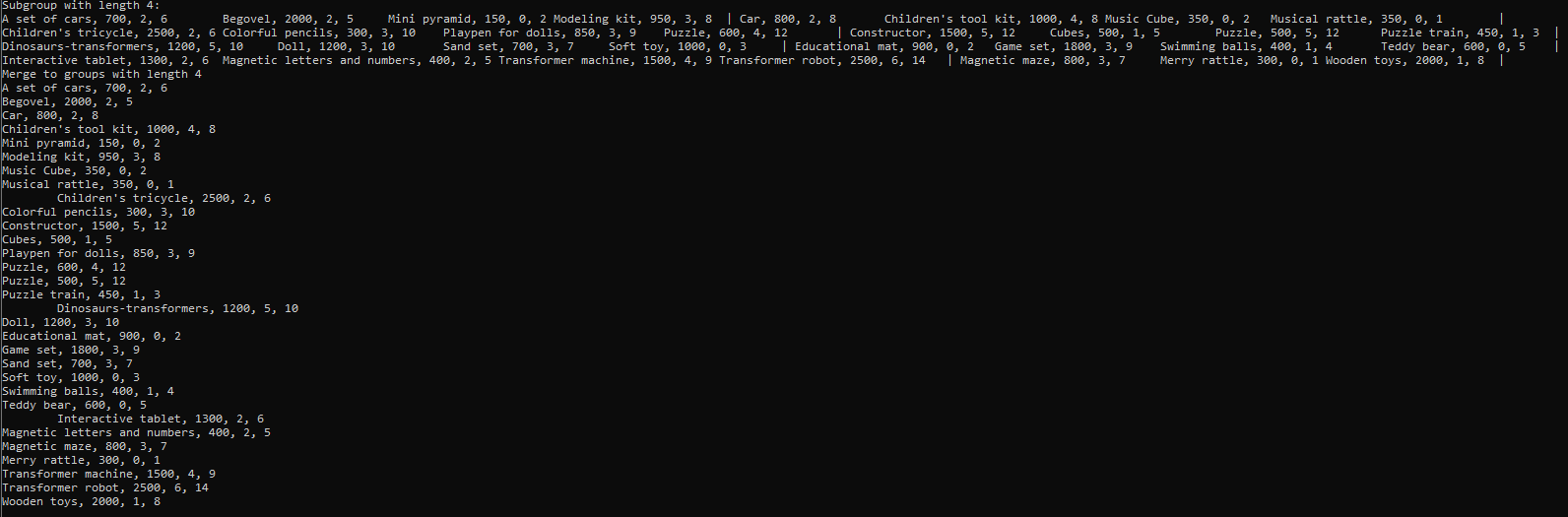


Рисунок 5 - Тестирование адаптированной программы

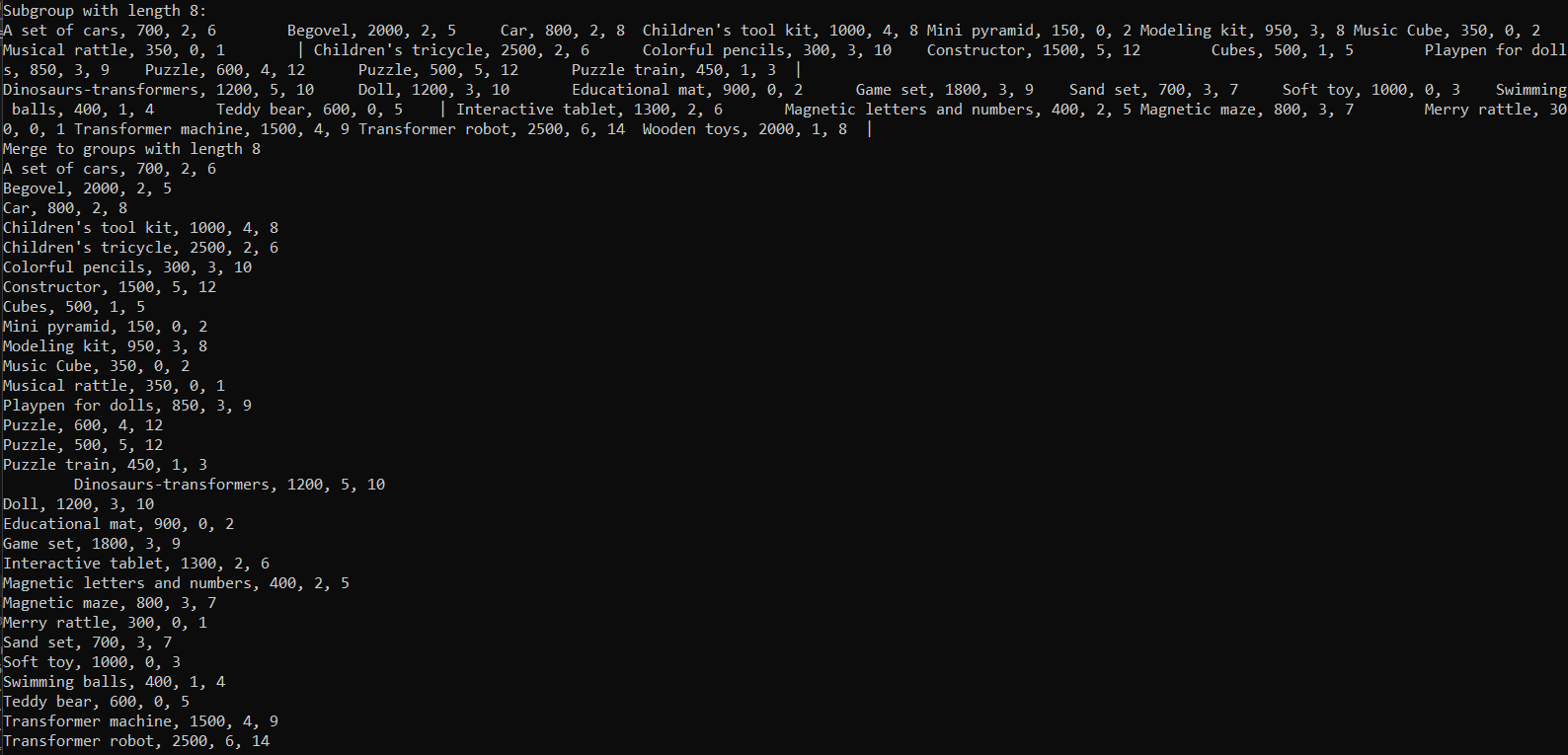


Рисунок 6 - Тестирование адаптированной программы

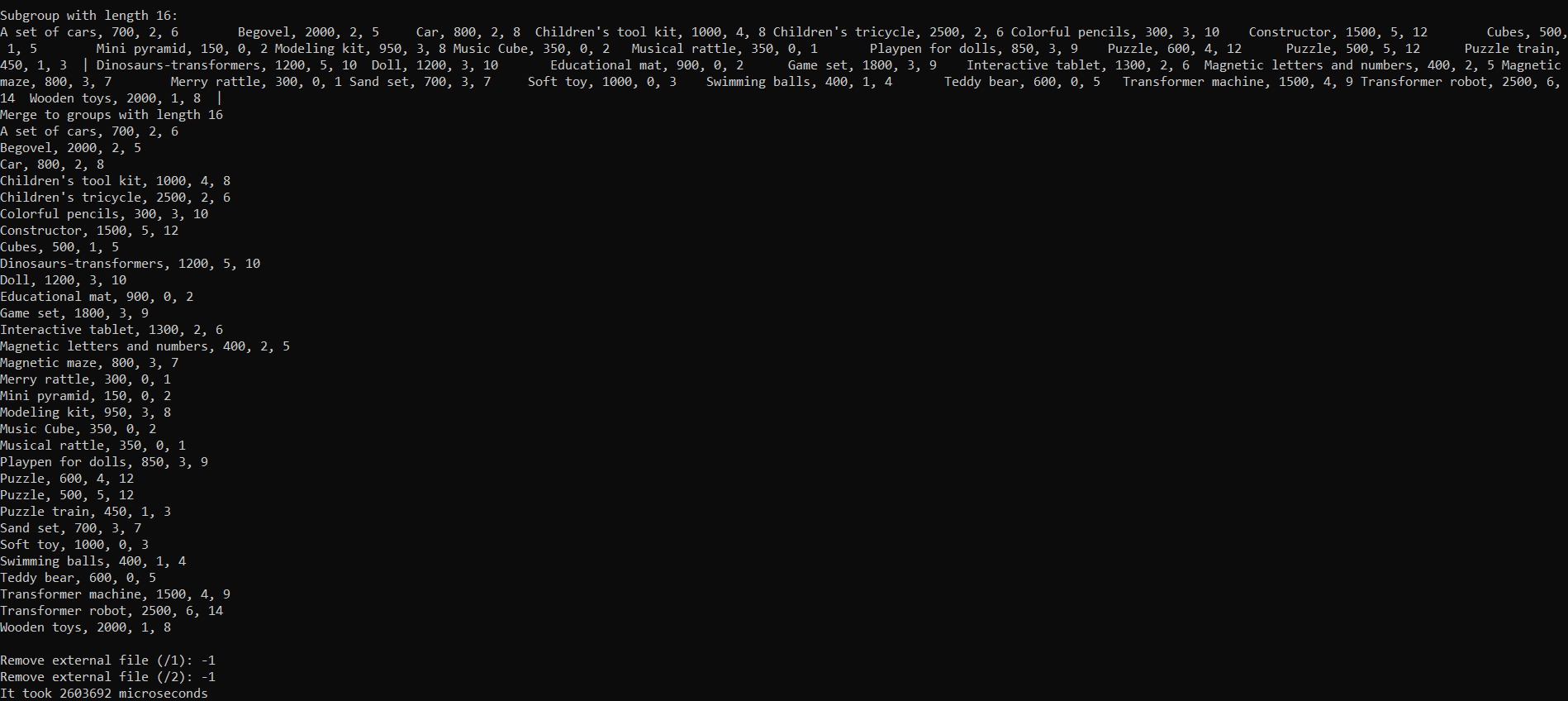


Рисунок 7 - Тестирование адаптированной программы

## **2.7 Практическая сложность алгоритма**

При увеличении количества записей в файле, увеличивается количество операций чтения и записи на диск, что сказывается на времени выполнения алгоритма. Докажем это утверждение с помощью тестирования на разном количестве записей(таб.1). Таким образом, практическая сложность алгоритма прямого слияния будет увеличиваться пропорционально увеличению количества записей в файле.

Таблица 1 - Сводная таблица результатов

| **Количество записей** | **Время(мкс)** |
| --- | --- |
| 8 | 203093 |
| 16 | 599637 |
| 32 | 2603692 |

## **2.8 Вывод по заданию №1**

Сортировка прямого слияния – это эффективный и стабильный алгоритм сортировки, который разделяет список на две части, сортирует их отдельно, а затем сливает отсортированные части в один упорядоченный список.

Этот алгоритм имеет сложность O(n log n) в любом случае, что делает его одним из самых эффективных в сортировке. Время выполнения зависит от количества элементов в списке, но гарантированно не превысит n log n операций. А ёмкостная сложность данного алгоритма равна O(n).

Прямое слияние требует дополнительной памяти для сортировки, так как создает временные массивы для слияния, поэтому не очень эффективен для больших объемов данных из-за потребления дополнительной памяти.Но при этом сортировка прямого слияния всё же более эффективная чем простые, быстрые или усовершенствованные сортировки, так как позволяет эффективно сортировать как большие, так и маленькие списки, что делает ее универсальным методом для различных задач.

# **3 ЗАДАНИЕ №2**

## **3.1 Формулировка задачи**

Разработать программу и применить алгоритм сортировки естественного слияния к сортировке файла с данными варианта (файл уже должен быть подготовлен в задании 1).

1) Реализовать функцию сортировки (возможно, с вспомогательными функциями) и основную подпрограмму main.

2) Отладить программу, протестировать на примере из п.4.

3) Адаптировать программу для сортировки файла с записями, протестировать на подготовленном ранее файле.

4) Сформировать таблицу результатов, указав количество записей и время сортировки.

## **3.2 Математическая модель решения алгоритма**

### **3.2.1 Описание выполнения и блок-схема алгоритма естественного слияния**

Сортировка естественного слияния (natural merge sort) - это способ сортировки данных, который использует принципы слияния (merge) для упорядочивания элементов. В этом алгоритме данные разделяются на отсортированные подсписки, которые затем сливаются вместе для получения отсортированного списка.

Основные шаги выполнения сортировки естественного слияния:

1. Разделение: исходный список делится на отсортированные подсписки.

2. Слияние: подсписки сливаются в большие отсортированные подсписки.

3. Повторение: процесс разделения и слияния продолжается до тех пор, пока не будет получен окончательно отсортированный список.

Преимущество сортировки естественного слияния состоит в том, что она эффективно сортирует данные, которые уже частично отсортированы. Кроме того, она не требует дополнительной памяти для хранения временных массивов, так как сортировка происходит на месте.

Для усовершенствования этой сортировки был предложен вариант предварительного разделения данных в файле на серии одной длины, загрузки каждой серии в оперативную память, сортировки этой серии, например, алгоритмом быстрой сортировки, и запись этих серий в исходный файл. Чем длиннее серию возможно выгрузить в память, отсортировать и вернуть в файл, тем эффективнее будет алгоритм самой сортировки.

Рассмотрим алгоритм и его фазы. Он так же является двухфазным.

1. Определить размер свободной оперативной памяти для выгрузки в нее серии из файла. В программе создаем массив для хранения серии buf.

2. Открыть исходный файл А, подлежащий сортировке.

3. Открыть два файла для записи В и С.

4. Считать последовательность данных в количестве достаточном для размещения в массиве buf. Отсортировать в массиве методом внутренней сортировки и записать в файл В.

5. Считать следующую последовательность данных в количестве достаточном для размещения в массиве buf. Отсортировать в массиве методом внутренней сортировки и записать в файл С.

6. Пункты 4 и 5 выполнять, пока все данные из файла А не будут переписаны отсортированными во вспомогательные файлы В и С.

7. Слить данные в файл А сначала из файла В, затем из файла С.

Теперь файл А содержит длинные упорядоченные серии, считаем, что данные в сериях упорядочены по возрастанию.

8. Фаза разделения включает поочередную запись серий из А в файлы В и С.

9. Фаза слияния имеет теперь следующий алгоритм:

• Считываем данные из одного и другого файлов, пока ai<ai+1, меньшее из сравниваемых записывать в файл А, пока одна из серий не будет исчерпана, тогда остаток другой переписываем в файл А, пока выполняется условие ai<ai+1.

• После этого считываем следующую серию и так пока один из файлов не станет пустым, тогда серии другого переписываются в файл А.

10. Пункты 8 и 9 повторяются пока в файл А, в результате слияния не будет переписана только одна серия.

Реализация данного описания выполнения алгоритма представлена в виде блок-схемы (рис.8).

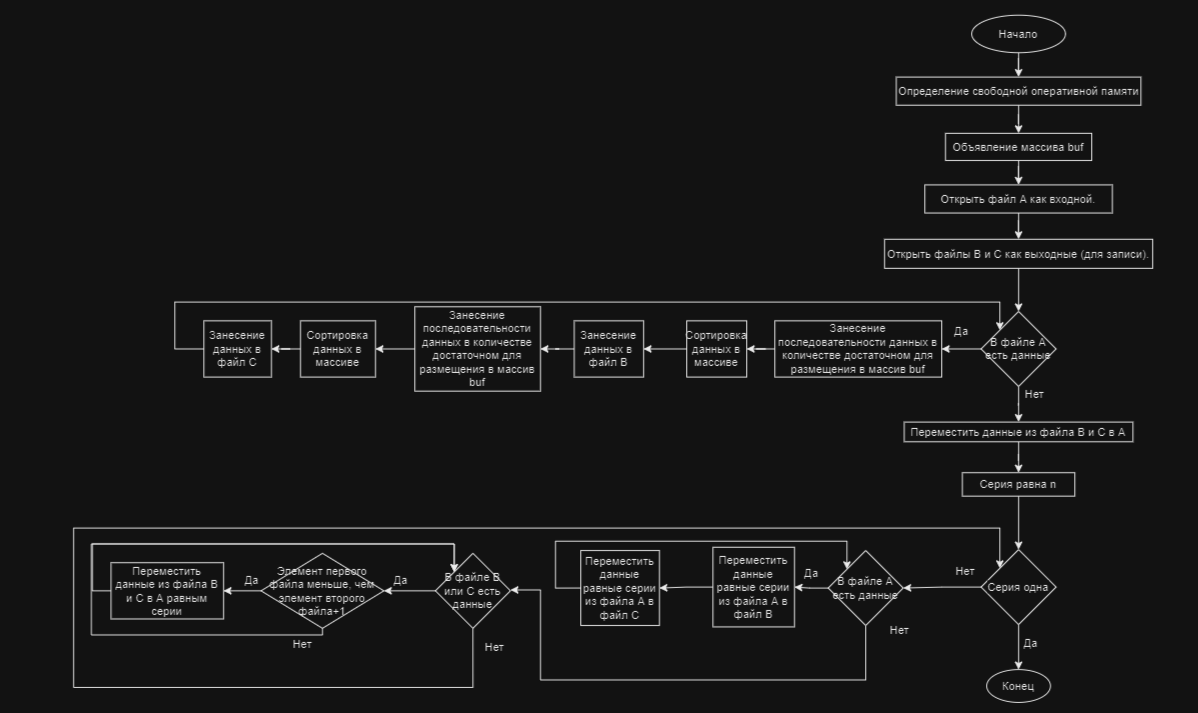


Рисунок 8 – Блок-схема алгоритма естественного слияния

### **3.2.2 Доказательство корректности циклов алгоритма естественного слияния**

Идея алгоритма естественного слияния заключается в разделении элементов массива на отсортированные подмассивы, с последующим слиянием их в отсортированные большие массивы. После каждого слияния размер подмассивов увеличивается, пока не будет получен один полностью отсортированный массив.

Для доказательства корректности циклов алгоритма естественного слияния через конечность циклов можно использовать метод индукции. Для каждого шага алгоритма можно показать, что процесс слияния приводит к уменьшению числа подмассивов и, следовательно, к их конечности.

Таким образом, применяя метод индукции, можно убедиться в корректности циклов алгоритма естественного слияния через доказанную конечность циклов.

### **3.2.3 Определение ситуаций лучшего, среднего и худшего случая и функции роста времени работы алгоритма естественного слияния**

Лучший случай: лучший случай для алгоритма естественного слияния возникает, когда входной массив уже частично отсортирован, то есть все подсписки являются отсортированными. В этом случае алгоритм сможет объединить эти частично отсортированные списки за один проход, что приведет к линейному времени выполнения.

Средний случай: средний случай для алгоритма естественного слияния включает в себя различные комбинации частично упорядоченных списков, что приведет к нескольким итерациям объединения и деления списков. В среднем алгоритм выполняется за O(n log n) времени, где n - количество элементов в списке.

Худший случай: худший случай для алгоритма естественного слияния возникает, когда массив содержит элементы в обратном порядке. В этом случае потребуется выполнить большее количество слияний и разделений, что приведет к наихудшему времени выполнения в O(n log n).

Функция роста времени работы алгоритма естественного слияния в среднем и худшем случаях составляет O(n log n), что является оптимальным временем выполнения для алгоритмов сортировки, работающих на сравнениях.

Ёмкостная сложность сортировки естественным слиянием составляет O(1).

## **3.3 Реализация алгоритма на языке C++ и проведение тестирования**

### **3.3.1 Реализация алгоритма естественного слияния на языке C++**

Реализуем данный алгоритм на языке C++(блок кода 1). Для реализации понадобятся такие библиотеки, как iostream, random, chrono, fstream, algorithm, vector, string.

iostream: библиотека в C++, являющаяся заголовочным файлом с классами, функциями и переменными для предоставления ввода и вывода данных через стандартные потоки cout (для вывода на консоль) и cin (для ввода с консоли).

chrono: библиотека в C++, используемая для работы с временем и временными измерениями, включая функции для измерения времени исполнения кода.

fstream: библиотека в C++, предоставляющая возможности для работы с файлами через потоки (ввода и вывода), позволяет записывать данные в файлы и читать данные из файлов.

algorithm: библиотека в C++, содержащая широкий набор стандартных алгоритмов, таких как сортировка, поиск, преобразование контейнеров и другие.

vector: библиотека в C++, представляющая собой динамический массив, позволяющий хранить и управлять элементами одного типа данных.

string: библиотека в C++, предоставляющая функции для работы с символьными строками, включая конкатенацию, поиск, сравнение и другие операции.

pragma once: это нестандартная, но широко поддерживаемая директива препроцессора, предназначенная для включения текущего заголовочного файла только один раз в одну компиляцию. Таким образом, #pragma когда-то служила той же цели, что и include guards, но с рядом преимуществ, включая меньшее количество кода, предотвращение конфликтов имен и иногда повышение скорости компиляции.

В программе также будут использоваться классы (в C++ — это абстракция, описывающая методы и свойства ещё не существующих объектов) и объекты (это конкретное представление абстракции, имеющее свои свойства и методы. Созданные объекты на основе одного класса называются экземплярами этого класса.)

| #include <iostream> #include <fstream> #include <algorithm> #include <vector> #include <string> #include <chrono> using namespace std; ostream& operator<< (ostream& lhv, const vector<int>& rhv) {  const auto size = rhv.size();  lhv << "(";  for (int i = 0; i < size; ++i)  {  lhv << rhv[i];  if(i+1 < size) lhv << " ";  }  lhv << ")";  return lhv; } constexpr auto kBlockSeparator = "|"; class NExternalMergeSort { public:  *// way\_count -- количество путей (вспомогательных файлов) сортировки*  explicit NExternalMergeSort(const string& filename, unsigned int way\_count = 2) : seriesLength\_{ 0 }, wayCount\_{ way\_count }, inputFilename\_{ filename } {  if (wayCount\_ < 2) wayCount\_ = 2;  };  virtual ~NExternalMergeSort() = default;  void sort(fstream& input); private:  *// вычисление длины серии*   void computeSeriesLength(fstream& input);  *// получить список из wayCount\_ вспомогательных файлов*  vector<fstream> getExternalFiles();  *// закрыть и открыть вспомогательные файлы для корректного ввода-вывода*  void reopenExternalFiles(vector<fstream>& ext, ios\_base::openmode mode);  *// удалить вспомогательные файлы*  void removeExternalFiles();  *// закрыть и открыть файл входных данных для корректного ввода-вывода*  void reopenInputFile(fstream& input, ios\_base::openmode mode);  bool fullEOF(vector<fstream>& ext);  *// проверка отсортированности входных данных*  bool isSorted(fstream& input);  *// разделение серии в сортированные подгруппы по порядку следования входных данных*  void split(fstream& input, vector<fstream>& ext);  *// естественное слияние из вспомогательных файлов*  void merge(fstream& input, vector<fstream>& ext);  unsigned int seriesLength\_; *// длина серии вычисляется при запуске сортировки*   unsigned int wayCount\_ = 2; *// количество путей (вспомогательных файлов) сортировки*   string inputFilename\_; }; int main(int argc, char\*\* argv) {  const auto input\_file = "A";  fstream input(input\_file, ios\_base::in | ios\_base::out);  if (!input)   {  perror("Error with input file");  return EXIT\_FAILURE;  }  NExternalMergeSort(input\_file).sort(input);  input.close();  system("pause");  return EXIT\_SUCCESS; } void NExternalMergeSort::computeSeriesLength(fstream& input\_series) {  this->seriesLength\_ = 0;  int value = 0;  while (input\_series >> value)  {  this->seriesLength\_++;  } } vector<fstream> NExternalMergeSort::getExternalFiles() {  vector<fstream> ext(this->wayCount\_);  for(unsigned int i = 0; i < this->wayCount\_; ++i)  {  const auto filename = to\_string(i + 1);  ofstream f(filename);  ext[i] = fstream(filename);  }  return ext; } void NExternalMergeSort::reopenExternalFiles(vector<fstream>& ext, ios\_base::openmode mode) {  for (int i = 0; i < ext.size(); ++i)  {  ext[i].close();  const auto filename = to\_string(i + 1);  ext[i].open(filename, mode);  } } void NExternalMergeSort::removeExternalFiles() {  for (unsigned int i = 0; i < this->wayCount\_; ++i)  {  const auto filename = to\_string(i + 1);  const auto removeResult = remove(filename.c\_str());  cout << "Remove external file (" << filename << "): " << boolalpha << removeResult << endl;  } } void NExternalMergeSort::reopenInputFile(fstream& input, ios\_base::openmode mode) {  input.close();  input.open(this->inputFilename\_, mode); } bool NExternalMergeSort::fullEOF(vector<fstream>& ext) {  bool fullEOF = true;  for (auto& f : ext)  {  fullEOF &= f.eof();  if (!fullEOF) break;  }  return fullEOF; } bool NExternalMergeSort::isSorted(fstream& input) {  int value = 0;  input >> value;  int oldValue = value;  bool sorted = false;  while (input >> value)  {  sorted = oldValue <= value;  oldValue = value;  if (!sorted) break;  }  return sorted; }  void NExternalMergeSort::split(fstream& input\_series, vector<fstream>& ext) {  int value = 0;  int oldValue = 0;  unsigned int way = 0;  bool firstStep = true;  while(input\_series >> value)  {  if (firstStep)  {  firstStep = false;   ext[way] << value << " ";  oldValue = value;  cout << value << " ";  continue;  }   if (oldValue > value)  {  ext[way].seekg(-1, ios\_base::cur);  ext[way] << kBlockSeparator << " ";  cout << kBlockSeparator;  way = (way + 1) % wayCount\_;  }  ext[way] << value << " ";  oldValue = value;  cout << value << " ";  }  cout << endl; } void NExternalMergeSort::merge(fstream& input\_series, vector<fstream>& ext) {  while(!this->fullEOF(ext))  {  vector<int> oldValues(this->wayCount\_, { 0 });  vector<bool> eogs(this->wayCount\_, { false }); *// end of group (аналог end of file)*  for (unsigned int way = 0; way < this->wayCount\_; ++way)  {  int value = 0;  if(ext[way] >> value) oldValues[way] = value;  else eogs[way] = true;  }  bool fullEOG = false; *// флаг достижения конца групп во всех путях*  *// выбираем путь, с которого будем начинать слияние*  while (!fullEOG)  {  int wayWithMin = -1;   for (unsigned int checkedWay = 0; checkedWay < this->wayCount\_; ++checkedWay)  {  if (eogs[checkedWay] || ext[checkedWay].eof()) continue;  wayWithMin = checkedWay;  }  if (-1 == wayWithMin) break;   for (unsigned int checkedWay = 0; checkedWay < this->wayCount\_; ++checkedWay)  {  if (eogs[checkedWay] || ext[checkedWay].eof()) continue;  if (oldValues[checkedWay] < oldValues[wayWithMin]) wayWithMin = checkedWay;  }  *// слияние*  input\_series << oldValues[wayWithMin] << " ";  cout << oldValues[wayWithMin] << " ";  *// след. значение*  int value = 0;  if (ext[wayWithMin] >> value)   {  oldValues[wayWithMin] = value;  }  else  {  eogs[wayWithMin] = true;  ext[wayWithMin].clear(); *// очищаем сигнальные биты потока, чтобы иметь возможность читать дальнейшие группы*  }  *// проверка достижения конца групп*  fullEOG = true;  for (auto eog : eogs)  {  fullEOG &= eog;  if (!fullEOG) break;  }  }  for (unsigned int way = 0; way < this->wayCount\_; ++way)  {  if (eogs[way])   {  ext[way].get();  ext[way].get();  }  }  }  cout << endl; } void NExternalMergeSort::sort(fstream& input\_series) {   this->computeSeriesLength(input\_series);  if (this->seriesLength\_ <= 1)  {  return;  }  vector<fstream> ext = this->getExternalFiles();  reopenInputFile(input\_series, ios\_base::in);  bool sorted = isSorted(input\_series);  while(!sorted)  {  reopenInputFile(input\_series, ios\_base::in);  reopenExternalFiles(ext, ios\_base::out);  cout << "Subgroup: " << endl;  this->split(input\_series, ext);  reopenInputFile(input\_series, ios\_base::out);  reopenExternalFiles(ext, ios\_base::in);  cout << "Merge: " << endl;  this->merge(input\_series, ext);  reopenInputFile(input\_series, ios\_base::in);  sorted = isSorted(input\_series);  cout << "Sorted: " << boolalpha << sorted << endl;  }  input\_series.flush();  this->removeExternalFiles(); } |
| --- |

Блок кода 4 – Программа main.cpp

### **2.3.2 Тестирование**

Стоит задача протестировать программу с заданным данными в файле А: 17 31 5 59 13 41 43 67 11 23 29 47 3 7 71 2 19 57 37 61. Выделим серии, завершая запятой, чтобы было нагляднее: 17 31’ 5 59’ 13 41 43 67’ 11 23 29 47’ 3 7 71’ 2 19 57’ 37 61. Получилось 7 серий. Разделим файл на два файла В и С, переписывая в них поочередно по серии: B: 17 31’ 13 41 43 67’ 3 7 71’ 37 61. C: 5 59’ 11 23 29 47’ 2 19 57. Сольем файлы в файл А, сливая серии в упорядоченные серии А:5 17 31 59’ 11 13 23 29 41 43 47 67’2 3 7 19 57 71’37 61. Опять разольем в В и С поочередно переписывая серии B: 5 17 31 59’2 3 7 19 57 71, C: 11 13 23 29 41 43 47 67’37 61. Сливаем в файл А по сериям А: 5 11 13 17 23 29 31 41 43 47 59 67’2 3 7 19 37 57 61 71. Разливаем ….. и продолжаем до тех пор, пока в массив А не будет переписана серия длины n

Воспользуемся структурой high\_resolution\_clock для подсчёта затраченного времени на сортировку. Для более точных результатов в программе будем рассматривать микросекунды.

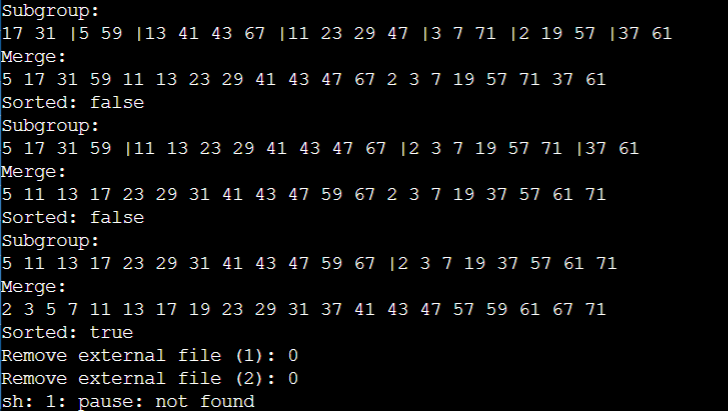


Рисунок 9 - Тестирование программы с входными данными из примера

## **3.4 Адаптация программы к новым данным**

Адаптируем программу к данным по условию индивидуального варианта. Магазин игрушек. Сведения об игрушке: Название (например: кукла, конструктор и т.д.), стоимость, возрастные границы детей (для кого игрушка предназначена) два поля – начальный возраст и конечный). То, что подчёркнуто, является ключом для сортировки. Зададим структуру файла, которая должна состоять из string (последовательности символов, хранящихся в массиве char) и трёх целочисленных значений.

Будет использоваться пользовательский заголовочный файл "ems\_natural.h" и библиотека functional (предоставляющая набор шаблонов классов для работы с функциональными объектами, а также набор вспомогательных классов для их использования в алгоритмах стандартной библиотеки.).

Реализация кода состоит из двух частей. Из заголовочного файла "ems\_natural.h"(блок кода 5) и исходного файла main.cpp(блок кода 6).

| #pragma once #include <iostream> #include <fstream> #include <algorithm> #include <vector> #include <string> #include <functional> using namespace std; *// SType - тип сортируемых данных* *// DSep - разделитель данных в файле* template<class SType, char DSep = ' '> class SExternalMergeSort { public:  *// way\_count -- количество путей (вспомогательных файлов) сортировки*  explicit NExternalMergeSort(const std::string& filename, unsigned int way\_count = 2) : seriesLength\_{ 0 }, wayCount\_{ way\_count }, inputFilename\_{ filename } {  if (wayCount\_ < 2) wayCount\_ = 2;  };  virtual ~NExternalMergeSort() = default;  template<class Compare>  void sort(fstream& input, Compare&& cmp = [](const SType& a, const SType& b) { return a < b; });  private:  *// вычисление длины серии*   void computeSeriesLength(std::fstream& input);  *// имя для вспомогательного файла*  string getExternalFilename(const unsigned int way);  *// получить список из wayCount\_ вспомогательных файлов*  vector<fstream> getExternalFiles();  *// закрыть и открыть вспомогательные файлы для корректного ввода-вывода*  void reopenExternalFiles(vector<fstream>& ext, ios\_base::openmode mode);  *// удалить вспомогательные файлы*  void removeExternalFiles();  *// закрыть и открыть файл входных данных для корректного ввода-вывода*  void reopenInputFile(fstream& input, ios\_base::openmode mode);  bool fullEOF(vector<fstream>& ext);  *// проверка отсортированности входных данных*  template<class Compare>  bool isSorted(std::fstream& input, Compare& cmp);  *// разделение серии в сортированные подгруппы по порядку следования входных данных*  template<class Compare>  void split(std::fstream& input, std::vector<std::fstream>& ext, Compare& cmp);  *// естественное слияние из вспомогательных файлов*  template<class Compare>  void merge(std::fstream& input, std::vector<std::fstream>& ext, Compare& cmp);  unsigned int seriesLength\_; *// длина серии вычисляется при запуске сортировки*   unsigned int wayCount\_ = 2; *// количество путей (вспомогательных файлов) сортировки*   std::string inputFilename\_; }; template<class SType, char DSep> *//Вычисляет длину серии данных, считывая их из файла input\_series до конца. Кол-во прочитанных элементов сохраняется в переменной*  seriesLength\_. void NExternalMergeSort<SType, DSep>::computeSeriesLength(fstream& input\_series) {  this->seriesLength\_ = 0;  SType value{};  while (input\_series >> value)  {  this->seriesLength\_++;  } } template<class SType, char DSep> *//Возвращает строку, представляющую внешний файл с данными, в зависимости от переданного параметра way. Файлы нумеруются с единицы.*  string NExternalMergeSort<SType, DSep>::getExternalFilename(const unsigned int way) {  return "/" + to\_string(way + 1); } template<class SType, char DSep> *//Создает и открывает вектор внешних файлов нужного размера и возвращает его. Названия файлов формируются с помощью функции getExternalFilename.*  vector<fstream> NExternalMergeSort<SType, DSep>::getExternalFiles() {  vector<fstream> ext(this->wayCount\_);  for (unsigned int i = 0; i < this->wayCount\_; ++i)  {  const auto filename = this->getExternalFilename(i);  ofstream f(filename);  ext[i] = fstream(filename);  }  return ext; } template<class SType, char DSep> *//Закрывает и повторно открывает все внешние файлы из вектора ext с заданным режимом mode. Названия файлов берутся из функции getExternalFilename.* void NExternalMergeSort<SType, DSep>::reopenExternalFiles(vector<fstream>& ext, ios\_base::openmode mode) {  for (int i = 0; i < ext.size(); ++i)  {  ext[i].close();   const auto filename = getExternalFilename(i);  ext[i].open(filename, mode);  } } template<class SType, char DSep> *//Удаляет внешние файлы, используемые для сортировки.* void NExternalMergeSort<SType, DSep>::removeExternalFiles() {  for (unsigned int i = 0; i < this->wayCount\_; ++i)  {  const auto filename = getExternalFilename(i);  const auto removeResult = remove(filename.c\_str());  cout << "Remove external file (" << filename << "): " << boolalpha << removeResult << endl;  } } template<class SType, char DSep> *//Повторно открывает входной файл с указанным режимом.* void NExternalMergeSort<SType, DSep>::reopenInputFile(fstream& input, ios\_base::openmode mode) {  input.close();  input.open(this->inputFilename\_, mode); }  template<class SType, char DSep> bool NExternalMergeSort<SType, DSep>::fullEOF(vector<fstream>& ext) {  bool fullEOF = true;  for (auto& f : ext)  {  fullEOF &= f.eof();  if (!fullEOF) break;  }   return fullEOF; } template<class SType, char DSep> template<class Compare> bool NExternalMergeSort<SType, DSep>::isSorted(fstream& input, Compare& cmp) {  SType value{};  input >> value;  SType oldValue = value;  bool sorted = false;  while (input >> value)  {  *// sorted = cmp(oldValue, value); // рабочий варик, если нет повторов*  *//sorted = cmp(oldValue, value);*  */\*const auto cmp1 = cmp(oldValue, value);  const auto cmp2 = cmp(value, oldValue);  sorted = (cmp1 == cmp2) || (cmp1 ^ !cmp2);\*/*  const auto cmp1 = cmp(oldValue, value);  const auto cmp2 = cmp(value, oldValue);  sorted = cmp1 || (cmp1 == false && cmp2 == false);   oldValue = value;  if (!sorted) break;  }   return sorted; } template<class SType, char DSep> template<class Compare> void NExternalMergeSort<SType, DSep>::split(  fstream& input\_series, vector<fstream>& ext,  Compare& cmp  ) {  SType value{};  SType oldValue{};  unsigned int way = 0;   if (input\_series >> value)  {  ext[way] << value << DSep;  oldValue = value;  cout << value << DSep;  }   while(input\_series >> value)  {  if (!cmp(oldValue, value) && cmp(value, oldValue))  {  way = (way + 1) % wayCount\_;  }   ext[way] << value << DSep;  oldValue = value;  cout << value << DSep;  }  cout << endl; }  template<class SType, char DSep> template<class Compare> void NExternalMergeSort<SType, DSep>::merge(  fstream& input\_series, vector<fstream>& ext,  Compare& cmp ) {  while(!this->fullEOF(ext))  {  vector<SType> oldValues(this->wayCount\_);  vector<bool> eogs(this->wayCount\_, { false }); *// end of group (аналог end of file)*  for (unsigned int way = 0; way < this->wayCount\_; ++way)  {  SType value{};  if(ext[way] >> value) oldValues[way] = value;  else eogs[way] = true;  }   bool fullEOG = false; *// флаг достижения конца групп во всех путях*   *// выбираем путь, с которого будем начинать слияние*  while (!fullEOG)  {  int wayWithMin = -1;   for (unsigned int checkedWay = 0; checkedWay < this->wayCount\_; ++checkedWay)  {  if (eogs[checkedWay] || ext[checkedWay].eof()) continue;  wayWithMin = checkedWay;  }  if (-1 == wayWithMin) break;   for (unsigned int checkedWay = 0; checkedWay < this->wayCount\_; ++checkedWay)  {  if (eogs[checkedWay] || ext[checkedWay].eof()) continue;  if (cmp(oldValues[checkedWay], oldValues[wayWithMin])) wayWithMin = checkedWay;  }   *// слияние*  input\_series << oldValues[wayWithMin] << DSep;  cout << oldValues[wayWithMin] << DSep;   *// след. значение*  SType value{};  if (ext[wayWithMin] >> value)   {  oldValues[wayWithMin] = value;  }  else  {  eogs[wayWithMin] = true;  ext[wayWithMin].clear(); *// очищаем сигнальные биты потока, чтобы иметь возможность читать дальнейшие группы*  }   *// проверка достижения конца групп*  fullEOG = true;  for (auto eog : eogs)  {  fullEOG &= eog;  if (!fullEOG) break;  }  }   *// переход через символы завершения группы*  for (unsigned int way = 0; way < this->wayCount\_; ++way)  {  if (eogs[way])   {  ext[way].get();  ext[way].get();  }  }  }   cout << endl; } template<class SType, char DSep> template<class Compare> void NExternalMergeSort<SType, DSep>::sort(fstream& input\_series, Compare&& cmp) {   this->computeSeriesLength(input\_series);  if (this->seriesLength\_ <= 1)  {  return;  }  auto&& cmp\_ = forward<Compare>(cmp);  vector<fstream> ext = this->getExternalFiles();  reopenInputFile(input\_series, ios\_base::in);  bool sorted = isSorted(input\_series, cmp\_);  cout << "Sorted: " << boolalpha << sorted << endl;  while(!sorted)  {  reopenInputFile(input\_series, ios\_base::in);  reopenExternalFiles(ext, ios\_base::out);  cout << "Subgroup: " << endl;  this->split(input\_series, ext, cmp\_);  reopenInputFile(input\_series, ios\_base::out);  reopenExternalFiles(ext, ios\_base::in);  cout << "Merge: " << endl;  this->merge(input\_series, ext, cmp\_);  reopenInputFile(input\_series, ios\_base::in);  sorted = isSorted(input\_series, cmp\_);  cout << "Sorted: " << boolalpha << sorted << endl;  }  input\_series.flush();  this->removeExternalFiles(); } |
| --- |

Блок кода 5 - заголовочный файл "ems\_natural.h"

| #include <fstream> #include <vector>  #include <chrono> *//Подключение пользовательского заголовочного файла "ems\_natural.h"* #include "ems\_natural.h" using namespace std; *//Определение структуры файла A, содержащего поля n, a, b, c.* struct A {  string n;*//Название игрушки*  int a = 0;*//цена игрушки*  int b = 0;*//возраст начальный*  int c = 0;*//возраст конечный* }; *//Перегрузка оператора >> для чтения значений типа* istream& operator>>(istream& lhv, A& rhv) {  string n;  while (true)  {  const auto c = lhv.get();  if (c == -1 || c == ',') break;  if (c >= 0) n += (char)c;  }  rhv.n = n;*// Присваиваем свойству "n" объекта "rhv" значение переменной "n" rhv.n = n*  int value = 0;*// Создаем переменную "value" и считываем значение из потока "lhv" int value = 0; lhv >> value;*  lhv >> value;*// Считываем значение а переменную "value" из потока "lhv" lhv >> value*  rhv.a = value;*// Присваиваем свойству "a" объекта "rhv" значение переменной "value", считанной из потока "lhv" rhv.a = value*  lhv.get();*// Пропускаем символ '\n' в потоке "lhv" lhv.get()*  lhv >> value;*// Снова считываем значение в переменную "value" из потока "lhv" lhv >> value*  rhv.b = value;*// Присваиваем свойству "b" объекта "rhv" значение переменной "value" rhv.b = value*  lhv.get();*// Пропускаем символ '\n' в потоке "lhv" lhv.get()*  lhv >> value;*// Снова считываем значение в переменную "value" из потока "lhv" lhv >> value*  rhv.c = value;*// Присваиваем свойству "c" объекта "rhv" значение переменной "value" rhv.c = value*  lhv.get(); *// считывание*  return lhv; } *//Перегрузка оператора << для записи значений типа A* ostream& operator<<(ostream& lhv, const A& rhv) {  lhv << rhv.n << ", " << rhv.a << ", " << rhv.b << ", " << rhv.c;  return lhv; } int main(int argc, char\*\* argv) {  setlocale(LC\_ALL, "RUS");  auto start = chrono::high\_resolution\_clock::now();*// Начало сортировки*  *//Открытие файла "A" в режиме чтения и записи.*  const auto input\_file = "A";  fstream input(input\_file, ios\_base::in | ios\_base::out);  *//Проверка успешности открытия файла*  if (!input)  {  perror("Error with input file");  return EXIT\_FAILURE;  }  *//Использование шаблонной функции NExternalMergeSort из заголовочного файла "ems\_simple.h"*  NExternalMergeSort<A, '\n'>(input\_file).sort(input, [](const A& a, const A& b) { return a.n < b.n; });  input.close();*//Закрытие входного файла*  auto end = chrono::high\_resolution\_clock::now();*// Конец сортировки*  auto d = chrono::duration\_cast<chrono::microseconds>(end - start).count();*// затраченное время*  cout << "It took " << d << " microseconds" << endl;  system("pause");*//Приостановка выполнения программы для ожидания ввода пользователя*   return EXIT\_SUCCESS;*//Возврат успешного завершения программы.* } |
| --- |

Блок кода 6 - Файл main.cpp

## **3.6 Тестирование адаптированного кода**

Стоит задача протестировать программу с заданным данными в файле А. Воспользуемся структурой high\_resolution\_clock для подсчёта затраченного времени на сортировку. Для более точных результатов в программе будем рассматривать микросекунды. Результаты тестирования будут продемонстрированы на рисунке 10-20.

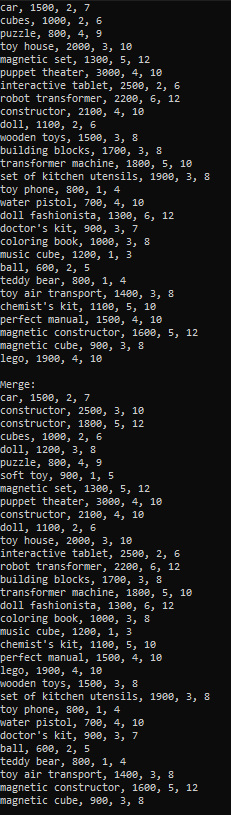


Рисунок 10 - Тестирование адаптированной программы

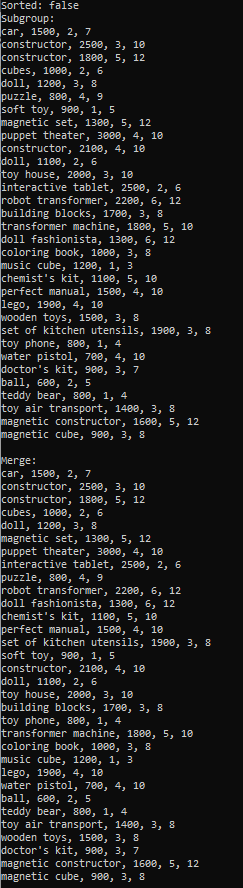


Рисунок 11 - Тестирование адаптированной программы

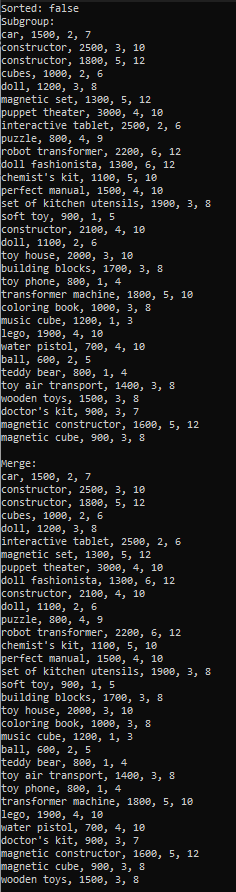


Рисунок 12 - Тестирование адаптированной программы

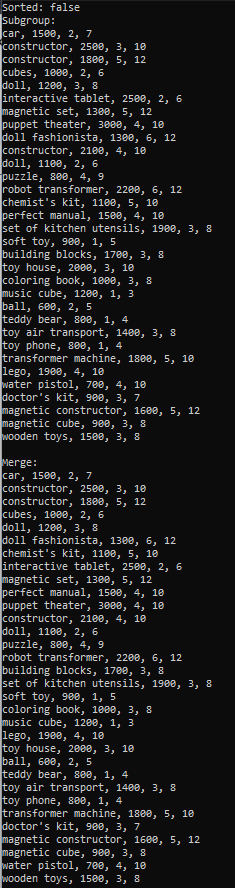


Рисунок 13 - Тестирование адаптированной программы

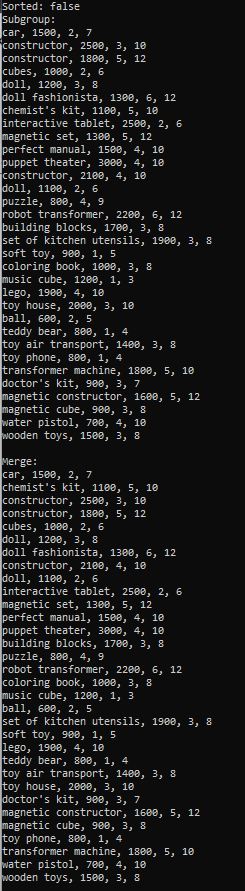


Рисунок 14 - Тестирование адаптированной программы

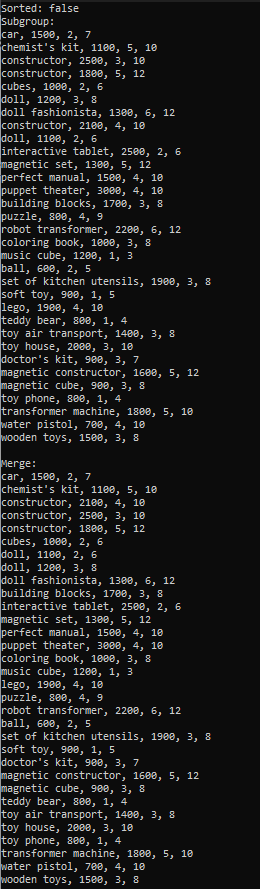


Рисунок 15 - Тестирование адаптированной программы

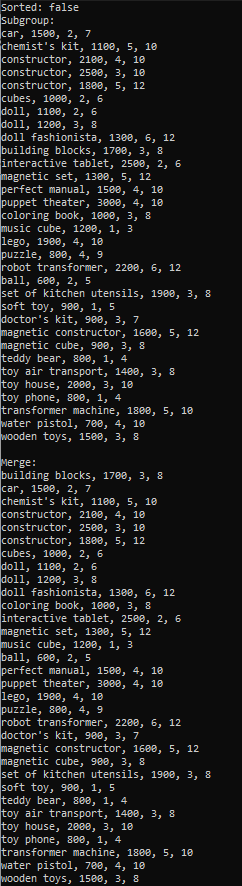


Рисунок 16 - Тестирование адаптированной программы

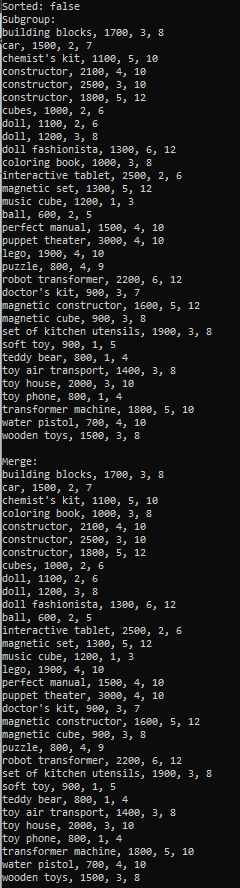


Рисунок 17 - Тестирование адаптированной программы

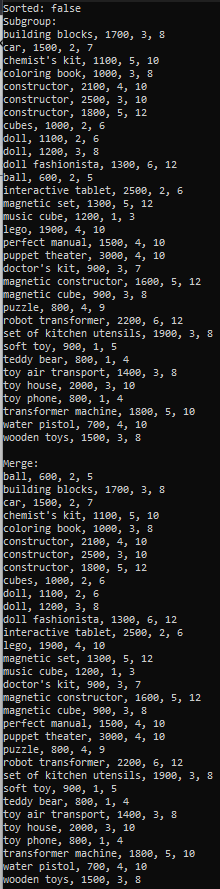


Рисунок 18 - Тестирование адаптированной программы

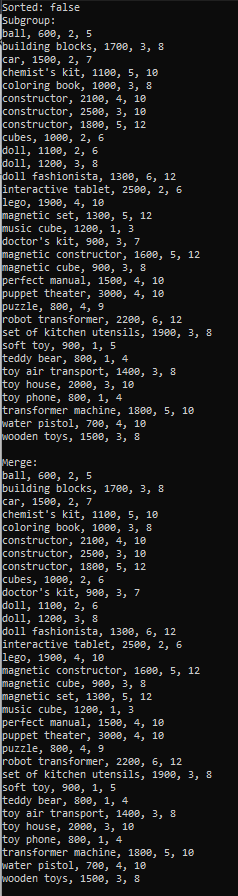


Рисунок 19 - Тестирование адаптированной программы



Рисунок 20 - Тестирование адаптированной программы

## **3.7 Практическая сложность алгоритма**

Для определения практической сложности алгоритма естественного слияния для файлов с увеличивающимся количеством записей, нужно учитывать количество операций сравнения и перемещения элементов в процессе сортировки. Докажем это утверждение с помощью тестирования на разном количестве записей(таб.2). Таким образом, практическая сложность алгоритма естественного слияния для файлов с увеличивающимся количеством записей остается примерно одинаковой и равна O(log n), где n - количество записей в файле.

Таблица 2 - Сводная таблица результатов

| **Количество записей** | **Время(мкс)** |
| --- | --- |
| 8 | 328906 |
| 16 | 902492 |
| 32 | 3130237 |

## **3.8 Вывод по заданию №2**

Сортировка естественного слияния - это алгоритм сортировки, который основан на разделении списка на отсортированные подсписки и их последующем слиянии. Он эффективен для сортировки больших списков, так как не требует дополнительной памяти для сортировки.

Практическая сложность сортировки естественного слияния зависит от количества элементов в списке. В лучшем случае сложность составляет O(n), а в худшем - O(n log n). Однако, в среднем случае сложность ближе к O(n log n). Ёмкостная сложность сортировки естественным слиянием составляет O(1).

Зависимость времени выполнения сортировки естественного слияния также зависит от количества элементов в списке. Чем больше элементов, тем больше времени потребуется для выполнения алгоритма. Но при этом он занимает меньше времени, чем классические алгоритмы сортировки, требующие дополнительной памяти.

Сортировка естественного слияния позволяет эффективно сортировать как большие, так и маленькие списки,что продемонстрировано в таблице 2, что делает ее универсальным методом для различных задач.

Сортировка естественным слиянием не требует дополнительной памяти для сортировки, так как использует принцип разделения и слияния подсписков, поэтому эффективна для больших списков и имеет стабильную сложность при увеличении количества записей.

В целом, сортировка естественным слиянием более эффективна и предпочтительна для сортировки больших списков, так как не требует дополнительной памяти и имеет стабильную практическую сложность. В то время как прямое слияние может быть более удобным для небольших списков, но может оказаться менее эффективным при работе с большими объемами данных из-за потребления дополнительной памяти.

# **5 ВЫВОДЫ**

В ходе практической работы были выполнены следующие задачи:

- Освоены приёмы сортировки данных из файлов;

- Проведён анализ алгоритмов прямого и естественного слияния;

- Были реализованы программы алгоритмов прямого и естественного слияния;

- Проведено тестирование программ для алгоритмов прямого и естественного слияния;

- Программы прямого и естественного слияния были адаптированы под индивидуальный вариант;

- Проведено тестирование программ для алгоритмов прямого и естественного слияния на разных количествах записей в файлах;

-Сделан вывод об эффективности данных алгоритмов;

Таким образом, главную цель практической работы, а именно освоение приёмов сортировки данных из файлов, можно считать выполненной.

# **6 ЛИТЕРАТУРА**

1. Бхаргава А. Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих. – СПб: Питер, 2017. – 288 с.

2. Вирт Н. Алгоритмы + структуры данных = программы. – М.: Мир, 1985. – 406 с.

3. Кнут Д.Э. Искусство программирования, том 3. Сортировка и поиск, 2-е изд. – М.: ООО «И.Д. Вильямс», 2018. – 832 с.

4. Кораблин Ю.П. Структуры и алгоритмы обработки данных: учебно-методическое пособие / Ю.П. Кораблин, В.П. Сыромятников, Л.А. Скворцова. – М.: РТУ МИРЭА, 2020. — 219 с.

5. Кормен Т.Х. и др. Алгоритмы: построение и анализ, 3-е изд. – М.: ООО «И.Д.Вильямс», 2013. – 1328 с.

6. Макконнелл Дж. Основы современных алгоритмов. Активный обучающий метод. 3-е доп. изд., - М.: Техносфера, 2018. – 416 с.

7. Седжвик Р. Фундаментальные алгоритмы на C++. Анализ/Структуры данных/Сортировка/Поиск. – К.: Издательство «Диасофт», 2001. – 688 с.

8. Скиена С. Алгоритмы. Руководство по разработке, - 2-е изд. – СПб: БХВ-Петербург, 2011. – 720 с.

9. Хайнеман Д. и др. Алгоритмы. Справочник с примерами на C, C++, Java и Python, 2-е изд. – СПб: ООО «Альфа-книга», 2017. – 432 с.

10. AlgoList – алгоритмы, методы, исходники [Электронный ресурс]. URL: http://algolist.manual.ru/ (дата обращения 15.03.2022).

11. Алгоритмы – всё об алгоритмах / Хабр [Электронный ресурс]. URL: https://habr.com/ru/hub/algorithms/ (дата обращения 15.03.2022).

12. НОУ ИНТУИТ | Технопарк Mail.ru Group: Алгоритмы и структуры данных [Электронный ресурс]. URL: https://intuit.ru/studies/courses/3496/738/info (дата обращения 15.03.2022).