**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра АМ**

Курсовая РАБОТА

**по дисциплине «Алгоритмы и структуры данных»**

Тема: Алгоритм сортировки Timsort

|  |  |  |
| --- | --- | --- |
| Студентка гр. 3385 |  | Дорогушина А.А. |
| Преподаватель |  | Жукова А.М. |

Санкт-Петербург

2024

**ЗАДАНИЕ**

**на курсовую работу**

|  |  |  |
| --- | --- | --- |
| Студентка Дорогушина А.А. | | |
| Группа 3385 | | |
| Тема работы: алгоритм сортировки Timsort | | |
|  | | |
| Содержание пояснительной записки:  Содержание, введение, функции, заключение, список использованных источников. | | |
| Предполагаемый объем пояснительной записки:  Не менее 20 страниц. | | |
| Дата выдачи задания: 29.11.2024 | | |
| Дата сдачи реферата: 19.12.2024 | | |
| Дата защиты реферата: 19.12.2024 | | |
| Студентка |  | Дорогушина А.А. |
| Преподаватель |  | Жукова А.М. |

**содержание**

|  |  |  |
| --- | --- | --- |
|  | Введение | 4 |
| 1. | Стандартные библиотеки C++ и другие | 5 |
| 2. | Функция считывания длины массива, необходимые структуры | 6 |
| 2.1. | Структура Pair | 6 |
| 2.2. | Считывание длины массива. Функция getValidLength | 6 |
| 3. | Описание функций проверок и вычислений | 7 |
| 3.1. | Функция isSorted | 7 |
| 3.2. | Функция getMinRun | 7 |
| 4. | Функции, организующие Timsort | 8 |
| 4.1. | Функция TimSort | 8 |
| 4.2. | Функция FindRun | 8 |
| 5. | Функции используемых сортировок | 9 |
| 5.1. | Функция InsertionSort и вспомогательная OutInsertionSortStep | 9 |
| 5.2. | Функция MergeSort и вспомогательная OutMergeSortStep | 10 |
| 6. | Функция main | 11 |
| 7. | Теория | 12 |
|  | Заключение | 16 |
|  | Список использованных источников | 17 |
|  | Приложение А. Исходный код программы | 18 |
|  | Приложение Б. Примеры работы программы | 25 |

**введение**

Цель работы: разработка программы на языке C++ с помощью библиотек iostream, vector и stack, работающей в зависимости от введённой длины массива, создавая, обрабатывая и выводя отсортированный массив рандомных чисел.

Для достижения поставленной цели требуется решить следующие задачи:

* реализовать необходимые структуры;
* реализовать функцию считывания длины массива;
* реализовать функцию проверки сортировки массива;
* реализовать функцию вычисления минимального размера упорядоченного подмассива;
* реализовать саму функцию тимсорт, вызывающую другие сортировки;
* реализовать функцию сортировки вставками;
* реализовать функцию сортировки слиянием;
* реализовать функцию объединения сортировок;
* реализовать пошаговую работу алгоритма в поток вывода.

**1. стандартные библиотеки C++ и другие**

Для корректной работы программы подключена стандартная библиотека языка C++ для ввода-вывода (*iostream*). А так же стандартная библиотека шаблонов, а именно последовательный контейнер и контейнер-адаптер соответственно (*vector* и *stack*)*.*

**2. Функция считывания длины массива, необходимые структуры**

**2.1. Структура Pair**

Эта структура определяет тип данных Pair, который содержит два целочисленных поля:

• *index*: для хранения индекса элемента или позиции в массиве;

• *size*: для хранения размера или длины.

Она пригодится при сортировке слиянием и работе со стеком.

**2.2. Считывание длины массива. Функция getValidLength**

Функция *getValidLength* предназначена для получения длины массива от пользователя. Она использует бесконечный цикл *while (true)*, чтобы продолжать запрашивать ввод, пока не будет получено корректное значение.

Cоздается переменная *valid*, которая изначально устанавливается в *true*. Затем программа проходит по каждому символу введенной строки *input*. Если хотя бы один символ не является цифрой *(!isdigit(c))*, то valid устанавливается в *false*, и цикл прерывается.

Если все символы в строке являются цифрами, программа пытается преобразовать строку в целое число с помощью функции *stoi*. Если полученное значение больше 1 миллиарда, выводится сообщение об ошибке. В противном случае функция возвращает корректную длину массива.

Если ввод оказался некорректным (например, содержал символы, отличные от цифр), пользователю выводится сообщение об ошибке, и программа снова запрашивает ввод.

**3. описание функий проверок и вычислений**

**3.1. Функция isSorted**

Эта функция проверяет, отсортирован ли массив arr в неубывающем порядке. Используется цикл for, который начинается с индекса 1 и проходит до конца массива. На каждой итерации функция сравнивает текущий элемент arr[i] с предыдущим элементом arr[i - 1]. Если обнаруживается, что предыдущий элемент больше текущего (arr[i - 1] > arr[i]), это означает, что массив не отсортирован, и функция возвращает false. Если цикл завершился без нахождения несоответствий, массив считается отсортированным, и функция возвращает true.

**3.2. Функция getMinRun**

Эта функция находит минимальную длину последовательности уже отсортированных элементов.

Переменная r: инициализируется как 0. Она будет использоваться для хранения информации о том, является ли длина массива четной или нечетной.

Цикл while: продолжается, пока n больше или равно 64. Внутри цикла используется операция n & 1, чтобы проверить, является ли n нечетным (если последний бит равен 1). Затем n сдвигается вправо на один бит (n >>= 1), что эквивалентно делению на 2.

После завершения цикла функция возвращает сумму n + r, которая будет находиться в диапазоне от 32 до 64.

**4. функции, организующие timsort**

**4.1. Функция TimSort**

Проверка размера: Если размер массива больше 63, вызывается функция FindRun, которая находит длину подмассивов, которые отсортирует сортировкой вставками, и сортирует их с помощью сортировки слиянием.

Сортировка вставками: Если размер массива 63 или меньше, используется сортировка вставками. Это связано с тем, что для небольших массивов сортировка вставками работает быстрее и проще.

**4.2. Функция FindRun**

Инициализация: Определяется размер массива n и длина подмассивов min\_run. Создается стек для хранения найденных подмассивов.

• Поиск конца подмассива: Внутренний цикл находит конец текущего подмассива для сортировки, если он не отсортирован.

• Сортировка вставками: Каждый найденный подмассив сортируется с помощью сортировки вставками и помещается в стек.

• Слияние: Когда в стеке больше одного "бега", они сливаются. Сначала извлекаются два верхних элемента из стека (run1 и run2).

• Слияние без третьего элемента: Если стек пуст после извлечения двух верхних элементов, выполняется слияние этих двух "бегов".

• Слияние с третьим элементом: Если в стеке есть третий элемент (run3), происходит более сложная логика слияния на основе размеров "бегов".

• Условия для слияния: Здесь реализована логика для определения порядка слияния "бегов", чтобы избежать неэффективных операций.

• Слияние на основе размеров: В зависимости от размеров "бегов" выполняется соответствующее слияние.

**5. функции используемых сортировок**

**5.1. Функция InsertionSort и вспомогательная OutInsertionSortStep**

Входные параметры:

• vector<int>& arr: массив, который нужно отсортировать;

• int l: левая граница (индекс начала сортировки);

• int r: правая граница (индекс конца сортировки);

• int len: длина массива.

Основной цикл проходит по элементам массива начиная со второго элемента (индекс l + 1) до последнего элемента (индекс r):

Сохраняется текущее значение элемента, который мы собираемся вставить на правильную позицию в отсортированной части массива.

Устанавливается индекс i на позицию перед j, чтобы начать сравнение с предыдущими элементами.

Начинается внутренний цикл, который сдвигает элементы массива вправо, пока не найдет правильное место для вставки значения value.

Если текущий элемент arr[i] больше чем value, он сдвигается на одну позицию вправо (arr[i + 1] = arr[i]).

После сдвига значение value вставляется на позицию arr[i].

Функция OutInsertionSortStep вызывается для вывода текущего состояния массива после каждого сдвига.

Функция OutInsertionSortStep:

Если длина массива len меньше или равна 10 проверяется условие: если это не последний шаг (j == len - 1) и (либо i < l, либо текущий элемент arr[i] меньше или равен value), то состояние массива выводится.

Если длина массива меньше 64, но больше 10 проверяется условие: если индекс i меньше l или текущий элемент arr[i] меньше или равен value, то состояние массива выводится.

Если длина больше, выведутся шаги только сортировки слиянием.

**5.2. Функция MergeSort и вспомогательная OutMergeSortStep**

Выводится текущее состояние массива перед началом сортировки.

Создаются подмассивы: Определяются размеры левой и правой частей массива. Создаются временные массивы left\_arr и right\_arr. Элементы копируются из исходного массива в соответствующие временные массивы.

Выполняется слияние подмассивов: Используются указатели для отслеживания текущих позиций в обоих подмассивах. Сравниваются элементы из левого и правого подмассивов: Меньший элемент добавляется в основной массив. Указатель на соответствующий подмассив увеличивается.

Каждый раз когда указатель увеличивается, вызывается OutMergeSortStep для вывода текущего состояния массива.

В функции предусмотрены условия, когда один из подмассивов может быть обработан полностью:

Если количество добавленных элементов из одного из подмассивов достигает 7, происходит дополнительная обработка оставшихся элементов из другого подмассива с использованием бинарного поиска для вставки.

Завершение слияния: После завершения основного цикла любые оставшиеся элементы из левого или правого подмассивов добавляются в основной массив. Каждый раз после добавления элемента вызывается OutMergeSortStep для вывода текущего состояния.

Функция OutMergeSortStep: вывод обработанной части массива.

**6. функция main**

В функции *main* происходит обработка входных аргументов командной строки, вывод обработанного массива и вызов проверки на правильность.

Создание рандомного массива: Пользователь вводит длину массива через функцию getValidLength(). С помощью srand(time(0)) и rand() % 1000 создается массив случайных целых чисел от 0 до 999.

Вывод исходного массива: Исходный массив выводится на экран, чтобы пользователь мог увидеть, что было до сортировки.

Сортировка: Вызывается функция TimSort(arr, len), которая сортирует массив. Предполагается, что эта функция реализует алгоритм TimSort, который является гибридом сортировки вставками и сортировки слиянием.

Вывод отсортированного массива: После сортировки массив снова выводится на экран, чтобы показать результат.

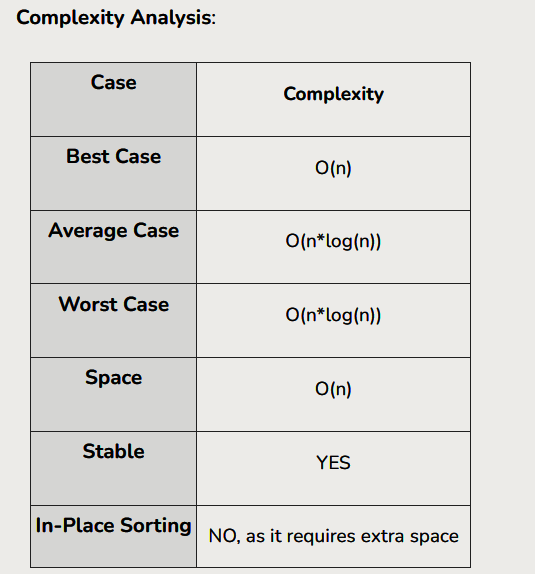
Проверка корректности сортировки: Используется функция isSorted(arr), чтобы проверить, правильно ли отсортирован массив. Если массив отсортирован верно, выводится соответствующее сообщение. Если нет — выводится сообщение об ошибке.

**7. теория**

Timsort: Очень быстрый, O(n log(n)), гибридный, стабильный алгоритм сортировки, созданный для реального мира, а не для академических целей. Тим Питерс создал Timsort для Python в 2001 году. Timsort официально реализован на C, а не на Python. Опубликован в 2002 году.

Timsort сначала анализирует список, который он пытается отсортировать, и на его основе выбирает наилучший подход. С момента его появления он используется в качестве алгоритма сортировки по умолчанию в Python и Java.

Время сортировки Timsort такое же, как и у Mergesort, что быстрее, чем у большинства других известных сортировок. Timsort фактически использует сортировку вставкой как и Mergesort.



Тим Питерс разработал Timsort для использования уже упорядоченных элементов, которые существуют в большинстве реальных наборов данных. Он называет эти упорядоченные элементы "natural runs" (естественные пробеги). Они итеративно перебирают данные, собирая элементы в пробеги и одновременно объединяя несколько этих пробегов в один.

Если массив, который мы пытаемся отсортировать, содержит менее 64 элементов, Timsort выполняет сортировку вставкой.

Сортировка вставкой — это простая сортировка, которая наиболее эффективна для небольших списков. Она функционирует довольно медленно при работе с большими списками, но при этом быстро работает с маленькими списками. Идея сортировки вставкой заключается в следующем:

* Просматривать элементы по одному
* Построить отсортированный список, вставляя элемент в нужное место

В данном случае мы вставляем только что отсортированные элементы в новый подмассив (sub-array), который начинается от начала массива.

Больше о пробегах

Если список больше 64 элементов, то алгоритм сделает первый проход по списку в поисках частей, которые строго возрастают или убывают. Если какая-либо часть является убывающей, то она будет изменена.

Minrun - это размер, который определяется на основе размера массива. Алгоритм выбирает его таким образом, чтобы большинство пробегов в произвольном массиве были или становились длиной minrun. Слияние двух массивов более эффективно, когда число пробегов равно или чуть меньше степени числа 2. Timsort выбирает minrun, чтобы попытаться обеспечить эту эффективность, убедившись, что minrun равен или меньше степени числа 2.

Алгоритм выбирает minrun из диапазона от 32 до 64 включительно, таким образом, чтобы наиболее эффективно работать с массивом.

Если длина пробега меньше minrun, необходимо рассчитать длину этого пробега относительно minrun. Используя это новое число выполняется сортировка вставкой для создания нового пробега.

Итак, если minrun равен 101, а длина пробега равна 51, вы выполняете вычитание: 101-51 = 50. Затем вы берете 51 элемент, и делаете сортировку вставкой для создания нового пробега.

После завершения этой части должно появиться множество отсортированных пробегов в списке.

Слияние

Теперь Timsort использует сортировку слиянием, чтобы объединить пробеги. Однако Timsort следит за тем, чтобы сохранить стабильность и баланс слияния во время сортировки.

Для поддержания стабильности мы не должны менять местами два одинаковых числа. Это не только сохраняет их исходные позиции в списке, но и позволяет алгоритму работать быстрее. В ближайшее время мы обсудим баланс слияния.

Когда Timsort находит пробеги, он добавляет их в стек. Простой стек выглядит следующим образом. Представьте себе стопку тарелок. Вы не можете взять тарелки снизу, поэтому вам приходится брать их сверху. То же самое можно сказать и о стеке.

Timsort пытается сбалансировать две конкурирующие задачи при сортировке слиянием. С одной стороны, мы хотели бы отложить слияние на как можно дольше, чтобы использовать паттерны, которые могут появиться позже, но еще больше нам хотелось бы произвести слияние как можно быстрее, чтобы воспользоваться тем, что только что найденный пробег все еще находится наверху в иерархии памяти. Мы также не можем откладывать слияние "слишком надолго", потому что на запоминание еще не объединенных пробегов расходуется память, а стек имеет фиксированный размер.

Чтобы добиться компромисса, Timsort отслеживает три последних элемента в стеке и создает два правила, которые должны выполняться для этих элементов:

1. A > B + C

2. B > C

Где A, B и C - три самых последних элемента в стеке.

По словам Тима Питерса:

Хорошим компромиссом оказалось то, что сохраняет два варианта для записей в стеке, где A, B и C - длины трех самых правых еще не объединенных частей.

Обычно объединить соседние пробеги разной длины очень сложно. Еще сложнее то, что мы должны поддерживать стабильность. Чтобы обойти эту проблему, Timsort выделяет временную память. Он помещает меньший (называя оба пробега A и B) из двух побегов в эту временную память.

Пока Timsort объединяет A и B, обнаруживается, что один пробег "выигрывает" много раз подряд. Если бы оказалось, что A состоял из гораздо меньших чисел, чем B, то A оказался бы на прежнем месте. Слияние этих двух пробегов потребовало бы много работы c нулевым результатом.

Чаще всего данные имеют некоторую уже существующую внутреннюю структуру. Timsort предполагает, что если многие значения пробега A меньше, чем значения пробега B, то, скорее всего, A будет и дальше содержать меньшие значения, чем B.

Затем Timsort переходит в режим galloping. Вместо того чтобы сверять A[0] и B[0] друг с другом, Timsort выполняет бинарный поиск соответствующей позиции b[0] в a[0]. Таким образом, Timsort может полностью переместить A. Затем Timsort ищет соответствующее местоположение A[0] в B. После этого Timsort также может полностью перемещает B.

Оказывается, что действие не имеет смысла, если подходящее место для B[0] находится очень близко к началу A (или наоборот), поэтому режим gallop быстро завершается, если он не приносит результатов.

Если говорить коротко, Timsort делает две вещи невероятно хорошо:

* Обеспечивает высокую производительность при работе с массивами с уже существующей внутренней структурой
* Создает возможность поддерживать стабильную сортировку

**заключение**

Была разработана программа на языке C++ с помощью библиотек iostream, vector и stack, работающая в зависимости от введённой длины массива. Реализовано создание, обработка и вывод отсортированного массива рандомных чисел.

Для достижения поставленной цели требуется были решены все необходимые задачи: реализованы необходимые структуры и функции.

Все цели были выполнены.

**список использованных источников**

1. https://www.geeksforgeeks.org/timsort/

2. https://habr.com/ru/companies/otus/articles/565640/

3. https://ru.wikipedia.org/wiki/Timsort

**приложение А**

**исходный код программы**

// инклюды и std для удобства

#include <iostream>

#include <vector>

#include <stack>

using namespace std;

// все функции

struct Pair; // мы с тамарой...

int getValidLength(); // получение длины

bool isSorted(vector<int>& arr); // проверка

int getMinRun(int n); // вычисление минрана

void TimSort(vector<int>& arr, int len); // сам тимсорт

void InsertionSort(vector<int>& arr, int l, int r, int len); // сортировка 1

void OutInsertionSortStep(vector<int>& arr, int len, int i, int j, int l, int value); // шажочек 1

void MergeSort(vector<int>& arr, int left, int mid, int right); // сортровка 2

void OutMergeSortStep(vector<int>& arr, int arr\_pointer); // шажочек 2

void FindRun(vector<int>& arr, int len); // объединение

// структура индекса и размера, пригодится при сортировке слиянием

struct Pair {

int index;

int size;

};

// получаем длину массива через поток ввода с проверкой на дурака

int getValidLength() {

string input;

int length = 0;

while (true) {

cout << "\nВведите длину массива: ";

getline(cin, input);

bool valid = true;

for (char c : input) {

if (!isdigit(c)) {

valid = false;

break;

}

}

if (valid) {

length = stoi(input);

if (length > 1000000000) {

cout << "Слишком большая величина" << endl;

}

else {

return length;

}

}

cout << "Ошибка: введите только цифры от 0 до 9.\n";

system("pause");

}

}

// проверка отсортированного массива

bool isSorted(vector<int>& arr) {

for (int i = 1; i < arr.size(); i++) {

if (arr[i - 1] > arr[i]) {

return false;

}

}

return true;

}

// нахождение минимального наибольшего упорядоченного подмассива (оптимально от 32 до 64)

int getMinRun(int n) {

int r = 0;

while (n >= 64) {

r |= (n & 1);

n >>= 1;

}

return n + r;

}

// сам тимсорт, если длина массива не превышает 63, даже не лезет в сортировку слиянием

void TimSort(vector<int>& arr, int len) {

if (arr.size() > 63) {

FindRun(arr, len);

}

else {

InsertionSort(arr, 0, arr.size() - 1, len);

}

}

// сортировка вставками с пошаговым выводом

void InsertionSort(vector<int>& arr, int l, int r, int len) {

for (int j = l + 1; j < r + 1; j++) {

int value = arr[j];

int i = j - 1;

while (i >= l && arr[i] > value) {

arr[i + 1] = arr[i];

arr[i] = value;

i--;

OutInsertionSortStep(arr, len, i, j, l, value);

}

}

}

// пошаговый вывод

void OutInsertionSortStep(vector<int>& arr, int len, int i, int j, int l, int value) {

if (len <= 10){

if (!(j == len-1 && (i < l | arr[i] <= value))){

cout << "\nШаг сортировки вставками " << j << ": ";

for (int i = 0; i < len; i++)

cout << arr[i] << " ";

}

if (i < l | arr[i] <= value)

cout << "\n";

}else if (len < 64){

if (i < l | arr[i] <= value){

cout << "\nШаг сортировки вставками " << j << ": ";

for (int i = 0; i < len; i++)

cout << arr[i] << " ";

cout << "\n";

}

}

}

// сортировка слиянием с пошаговым выводом

void MergeSort(vector<int>& arr, int left, int mid, int right) {

cout << "\nНачало сортировки слиянием: ";

for (int i = 0; i < arr.size(); i++)

cout << arr[i] << " ";

cout << "\n";

int left\_size = mid - left + 1;

int right\_size = right - mid;

vector <int> left\_arr(left\_size);

vector <int> right\_arr(right\_size);

for (int i = mid + 1, j = 0; i <= right; i++, j++) {

right\_arr[j] = arr[i];

}

for (int i = left, j = 0; i <= mid; i++, j++) {

left\_arr[j] = arr[i];

}

int left\_pointer = 0;

int right\_pointer = 0;

int arr\_pointer = left;

int left\_counter = 0;

int right\_counter = 0;

while (left\_pointer < left\_arr.size() && right\_pointer < right\_arr.size())

{

if (left\_arr[left\_pointer] <= right\_arr[right\_pointer])

{

arr[arr\_pointer++] = left\_arr[left\_pointer++];

left\_counter++;

right\_counter = 0;

OutMergeSortStep(arr, arr\_pointer);

}

else

{

arr[arr\_pointer++] = right\_arr[right\_pointer++];

right\_counter++;

left\_counter = 0;

OutMergeSortStep(arr, arr\_pointer);

}

if (left\_counter == 7) {

while (right\_pointer < right\_arr.size() && left\_pointer < left\_arr.size()) {

int target = right\_arr[right\_pointer];

int low = left\_pointer;

int high = left\_arr.size();

while (low < high) {

int mid = low + (high - low) / 2;

if (left\_arr[mid] < target) {

low = mid + 1;

}

else {

high = mid;

}

}

while (left\_pointer < low) {

arr[arr\_pointer++] = left\_arr[left\_pointer++];

OutMergeSortStep(arr, arr\_pointer);

}

arr[arr\_pointer++] = target;

right\_pointer++;

OutMergeSortStep(arr, arr\_pointer);

}

}

else if (right\_counter == 7) {

while (left\_pointer < left\_arr.size() && right\_pointer < right\_arr.size()) {

int target = left\_arr[left\_pointer];

int low = right\_pointer;

int high = right\_arr.size();

while (low < high) {

int mid = low + (high - low) / 2;

if (right\_arr[mid] < target) {

low = mid + 1;

}

else {

high = mid;

}

}

while (right\_pointer < low) {

arr[arr\_pointer++] = right\_arr[right\_pointer++];

OutMergeSortStep(arr, arr\_pointer);

}

arr[arr\_pointer++] = target;

left\_pointer++;

OutMergeSortStep(arr, arr\_pointer);

}

}

}

while (left\_pointer < left\_size) {

arr[arr\_pointer++] = left\_arr[left\_pointer++];

OutMergeSortStep(arr, arr\_pointer);

}

while (right\_pointer < right\_size) {

arr[arr\_pointer++] = right\_arr[right\_pointer++];

OutMergeSortStep(arr, arr\_pointer);

}

}

// пошаговый вывод

void OutMergeSortStep(vector<int>& arr, int arr\_pointer) {

if (arr\_pointer < arr.size()){

cout << "\nШаг сортировки слиянием: ";

for (int i = 0; i < arr\_pointer; i++)

cout << arr[i] << " ";

cout << "\n";

}

};

// совместная работа сортировок с помощью стека (при длине массива большей 63)

void FindRun(vector<int>& arr, int len) {

int n = arr.size();

int min\_run = getMinRun(n);

int start = 0;

stack <Pair> run\_stack;

while (start < n) {

int end = start;

while (end + 1 < n && arr[end] <= arr[end + 1]) {

end++;

}

if (end + 1 < n && arr[end] > arr[end + 1]) {

while (end + 1 < n && arr[end] > arr[end + 1]) {

end++;

}

int left = start;

int right = end;

while (left < right) {

int temp = arr[left];

arr[left] = arr[right];

arr[right] = temp;

left++;

right--;

}

}

if (end - start + 1 < min\_run) {

if (start + min\_run - 1 < n) {

end = start + min\_run - 1;

}

else {

end = n - 1;

}

}

InsertionSort(arr, start, end, len);

run\_stack.push({ start, end - start + 1 });

start = end + 1;

}

while (run\_stack.size() > 1) {

Pair run1 = run\_stack.top();

run\_stack.pop();

Pair run2 = run\_stack.top();

run\_stack.pop();

if (run\_stack.empty()) {

MergeSort(arr, run2.index, run2.index + run2.size - 1, run1.index + run1.size - 1);

run\_stack.push({ run2.index, run2.size + run1.size });

}

else {

Pair run3 = run\_stack.top();

run\_stack.pop();

int X = run1.size;

int Y = run2.size;

int Z = run3.size;

if (Z > Y + X && Y > X) {

run\_stack.push(run3);

run\_stack.push(run2);

run\_stack.push(run1);

break;

}

if (X <= Z) {

MergeSort(arr, run3.index, run3.index + run3.size - 1, run2.index + run2.size - 1);

run\_stack.push({ run3.index, run3.size + run2.size });

run\_stack.push(run1);

}

else {

MergeSort(arr, run2.index, run2.index + run2.size - 1, run1.index + run1.size - 1);

run\_stack.push({ run2.index, run2.size + run1.size });

run\_stack.push(run3);

}

}

}

}

// создание рандомного массива, вывод исходника, вызов тимсорта, вывод результата и его проверка

int main() {

vector<int> arr;

int len;

len = getValidLength();

srand(time(0));

for (int i = 0; i < len; i++)

{

arr.push\_back(rand() % 1000);

}

cout << "\nИсходный массив: ";

for (int i = 0; i < len; i++)

cout << arr[i] << " ";

cout << "\n";

TimSort(arr, len);

cout << endl << "Отсортированный массив: ";

for (int i = 0; i < len; i++)

cout << arr[i] << " ";

cout << "\n";

if (isSorted(arr)) {

cout << endl << "Массив отсортирован верно\n";

}

if (!isSorted(arr)) {

cout << "Массив отсортирован не верно\n";

}

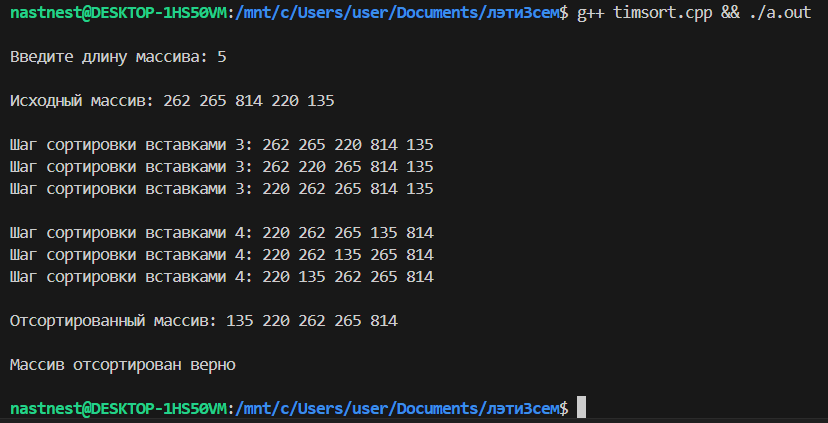
cout << endl;

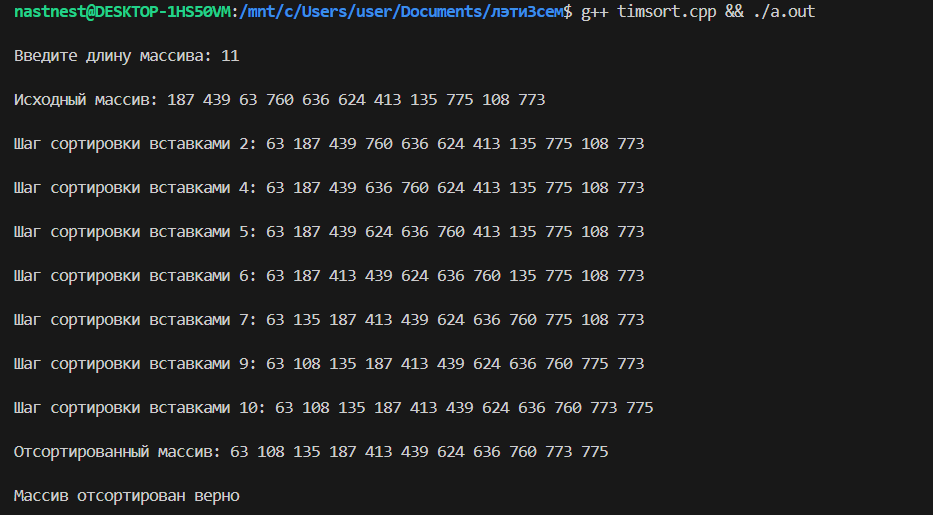
return 0;

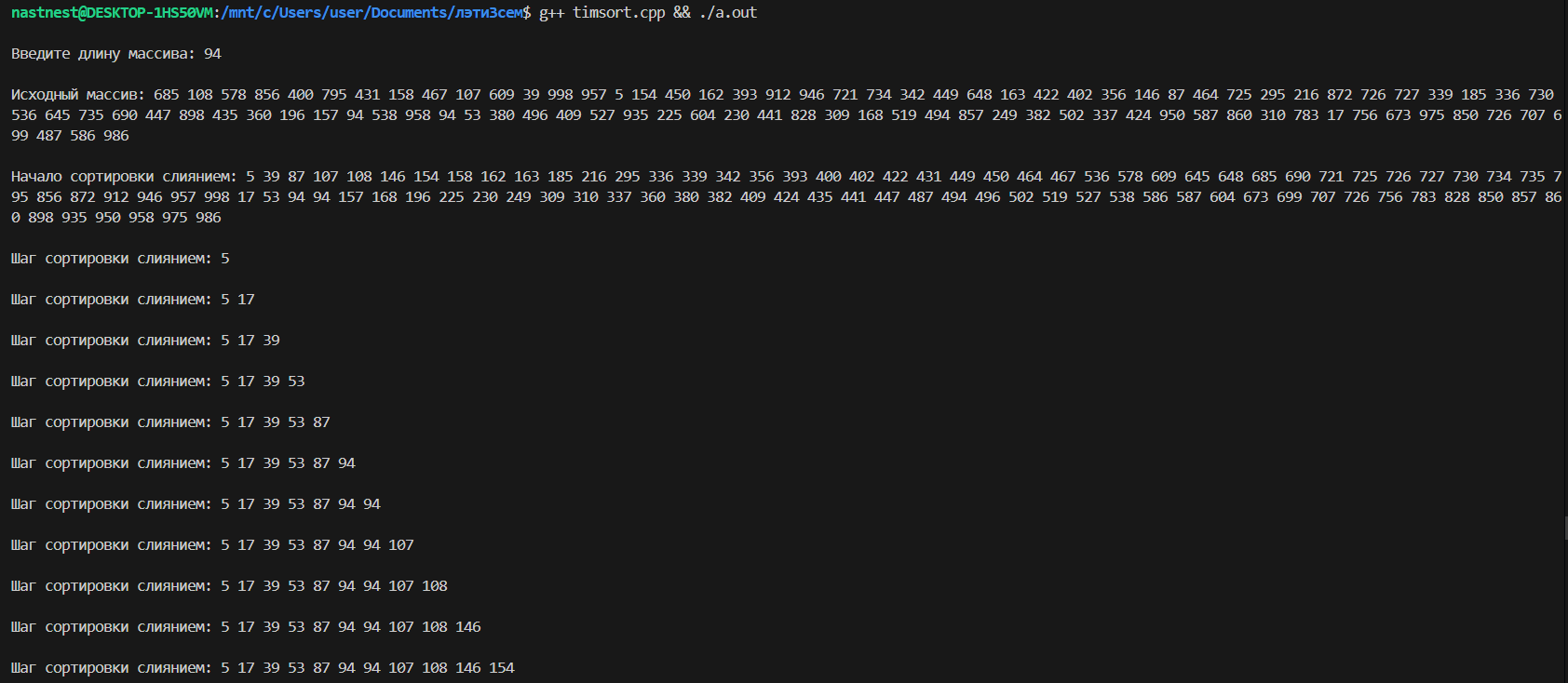
}

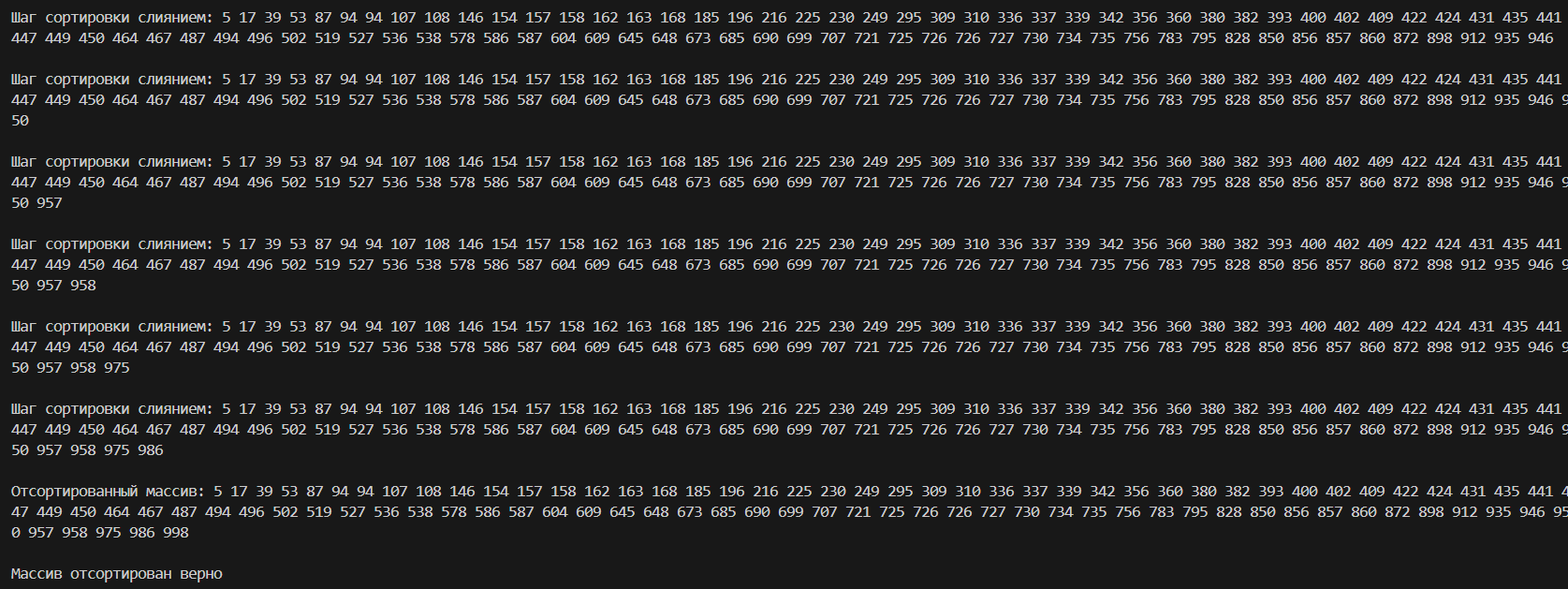
**приложение Б**

**примеры работы программы**

1. Тестирование функции сортировки вставками (длина <10)

2. Тестирование функции сортировки вставками (10 < длина < 64)

3. Тестирование функции сортировки слиянием (длина >63)



4. Тестирование ошибок

