ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 5

«Алгоритмы сортировки»

Выполнил работу

Васькин Виктор

Академическая группа № J3110

Принято

Вершинин Владислав Константинович

Санкт-Петербург

2024

**Структура отчёта:**

1. Введение

Цель данной лабораторной работы — сравнить три различных алгоритма сортировки: сортировку коктейльным встряхиванием (Cocktail Shaker Sort), сортировку кучей (Heap Sort) и сортировку по карманам (Pigeonhole Sort). Для этого необходимо протестировать каждый алгоритм на массиве данных и измерить время выполнения сортировки.

Для достижения указанной цели лабораторной работы необходимо решить несколько задач:

1. Изучить каждый из алгоритмов сортировки, их принципы работы и применимость в зависимости от характеристик данных.
2. Реализовать функции сортировки на языке C++ для каждого из выбранных алгоритмов.
3. Провести тестирование разработанных алгоритмов на массивах разных размеров (от 1000 до 10000 элементов) и различными распределениями значений.
4. Записать время выполнения каждого алгоритма в файлы, чтобы можно было проанализировать производительность.
5. Построить графические зависимости времени выполнения алгоритмов от длины массива, чтобы визуально оценить их эффективность.
6. Проанализировать временные и пространственные характеристики каждого алгоритма, проведя сравнение их сложности.
7. Проанализировать соответствие теоретической асимптотики с практическими результатами, выяснить наличие выбросов в данных и их причины, а также определить, в каких случаях целесообразно использовать каждый из рассматриваемых алгоритмов.
8. Теоретическая подготовка

2.1. Основные понятия и определения

**Сортировка** — процесс упорядочивания элементов в массиве или векторе по определённому критерию. Существуют различные алгоритмы сортировки, каждый из которых имеет свои преимущества и недостатки в зависимости от типа данных и их распределения.

2.2. Сортировка коктейльным встряхиванием (Cocktail Shaker Sort)

**Cocktail Shaker Sort** — это усовершенствованная версия сортировки пузырьком, которая проходит по массиву в двух направлениях: сначала от начала до конца, затем от конца к началу. Это позволяет эффективно перемещать не только большие, но и маленькие элементы к границам массива.

Преимущества:

* Простой в реализации и понимании.
* Может работать быстрее, чем классическая сортировка пузырьком на частично отсортированных массивах.

2.3. Сортировка кучей (Heap Sort)

**Heap Sort** — алгоритм сортировки, основанный на структуре данных "куча". Он строит max-heap из входных данных и затем извлекает наибольший элемент по одному, помещая его в отсортированный массив.

Преимущества:

* Временная сложность O(n log n), что делает его эффективным для больших массивов.
* Работает на месте без дополнительных массивов.

2.4. Сортировка по карманам (Pigeonhole Sort)

**Pigeonhole Sort** — это алгоритм сортировки, который особенно эффективен, когда диапазон значений элементов известен и мал по сравнению с их количеством. Он создает "карманы" для элементов в зависимости от диапазона, а затем собирает их в отсортированном порядке.

Преимущества:

* Временная сложность O(n + range), что делает его быстрым для определённых условий.

2.5. Временная и пространственная сложность

Cocktail Shaker Sort:

* Временная сложность:
* Худший случай: O(n^2) — при обратном порядке элементов.
* Лучший случай: O(n) — при уже отсортированном массиве.
* Пространственная сложность: O(1) — используется фиксированное количество вспомогательных переменных.

Heap Sort:

* Временная сложность: O(n log n) — для всех случаев.
* Пространственная сложность: O(1) — не использует дополнительных массивов.

Pigeonhole Sort:

* Временная сложность: O(n + range) — для нахождения минимума и максимума.
* Пространственная сложность: O(range) — выделяет память в зависимости от диапазона значений.

2.6. Понимание алгоритмов сортировки

Для успешного анализа и выбора алгоритма сортировки важно учитывать:

* Структуру данных и их характер (например, случайные, уже отсортированные, или сильно упорядоченные).
* Ожидаемый размер и диапазон значений данных, особенно для алгоритмов таких как Pigeonhole Sort, где диапазон может повлиять на память.

1. Реализация

3.1. Выбор инструментов

Для реализации задач по сортировке были использованы стандартные библиотеки C++:

* <vector> для хранения массивов данных и результатов сортировки.
* <algorithm> для использования функций, таких как std::swap, std::min\_element, и std::max\_element, которые помогают в упрощении кода.
* <fstream> для работы с файлами, что позволяет считывать данные и записывать результаты.

3.2. Разработка алгоритма

Процесс разработки алгоритмов состоял из нескольких ключевых этапов:

1. **Анализ задачи**: важно было понять, как каждый алгоритм работает и какие ситуации могут повлиять на их производительность. Были изучены основные принципы и временные характеристики каждого алгоритма.
2. **Определение структуры данных**: Для хранения промежуточных данных и считываемых значений были выбраны векторы (**std::vector**), которые позволяют динамически изменять размер и обеспечивают эффективное управление памятью.
3. **Написание основного кода**:

**3.2.1. Сортировка коктейльным встряхиванием (Cocktail Shaker Sort)**

**Алгоритм**:

* Представляет собой модификацию сортировки пузырьком, проходящую по массиву в двух направлениях — сначала слева направо, затем справа налево.
* Если за проход не произошло ни одного обмена, то массив считается отсортированным.

**Код**:

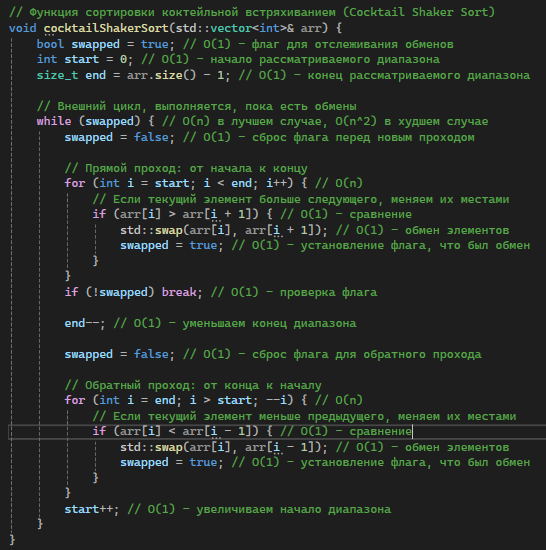


Рис. 1 CocktailShakerSort

**Анализ**:

* **Временная сложность**:
* Лучший случай: O(n) — если массив уже отсортирован.
* Средний случай: O(n^2) — при случайном порядке элементов.
* Худший случай: O(n^2) — если массив отсортирован в обратном порядке.
* **Пространственная сложность:** O(1) — используется только фиксированный набор переменных.
* **Применение:** подходит для небольших массивов и множества уже отсортированных элементов**.**

**3.2.2. Сортировка кучей (Heap Sort)**

**Алгоритм**:

* Строит max-heap и регулярно извлекает наибольший элемент, помещая его в конец массива.
* Использует метод heapify для поддержки структуры кучи.

**Код**:

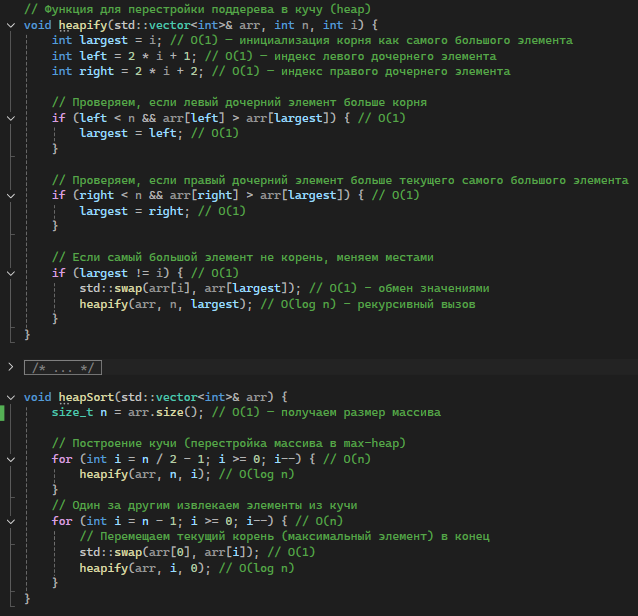


Рис. 2 heapSort

**Анализ**:

* **Временная сложность**:
* Лучший случай: O(n log n) — при любом расположении элементов.
* Средний случай: O(n log n) — для случайного порядка.
* Худший случай: O(n log n) — при любом порядке.
* **Пространственная сложность:** O(1) — не используется дополнительных массивов, алгоритм работает на месте.
* **Применение:** подходит для больших массивов и гарантирует стабильную производительность.

**3.2.3. Сортировка по карманам (Pigeonhole Sort)**

**Алгоритм**:

* Эффективен при известном диапазоне значений. Создает "карманы" для элементов в зависимости от их значений.
* После распределения элементов по карманам собирает их обратно в отсортированном порядке.

**Код**:

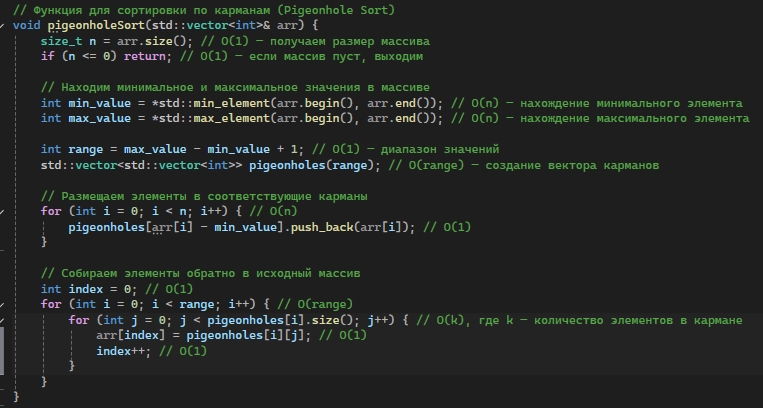


Рис. 3 PigeonholeSort

**Анализ**:

* **Временная сложность**:
* O(n) для нахождения минимального и максимального элементов.
* O(n + range) для размещения и сбора.
* Общая сложность: O(n + range).
* **Пространственная сложность:** O(range) — зависит от диапазона значений, используемого для карманов.
* **Применение:** пригоден для сортировки данных с ограниченным диапазоном значений, может быть неэффективным для больших диапазонов.

1. Экспериментальная часть

В ходе лабораторной работы было собрано большое количество данных, отражающих производительность трех различных алгоритмов сортировки: сортировки коктейльным встряхиванием (Cocktail Shaker Sort), сортировки кучей (Heap Sort) и сортировки по карманам (Pigeonhole Sort). На основе этих данных были построены графики, которые иллюстрируют зависимость времени выполнения от длины массивов. Эти графики позволяют проанализировать эффективность каждого алгоритма при различных условиях и оценить их поведение в лучшем, среднем и худшем случаях.

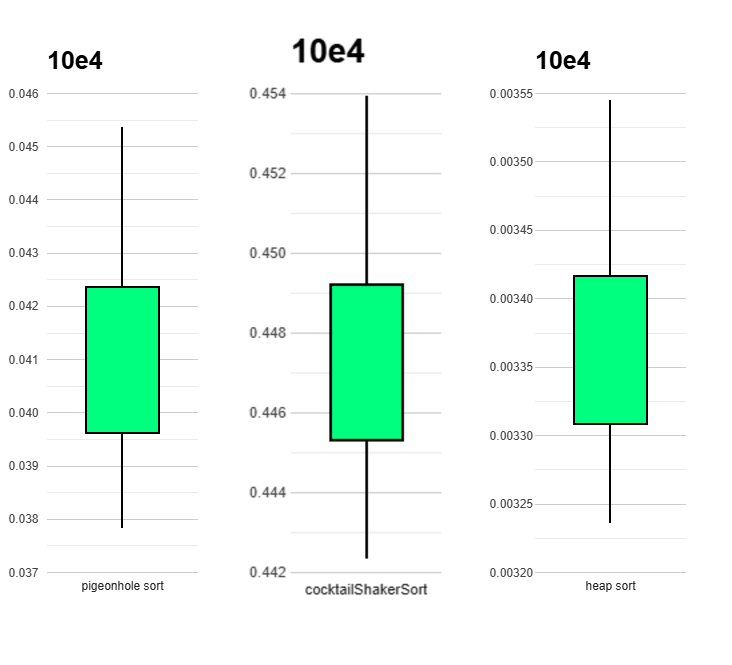


Рис. 4 График времени для длины массива 10е4

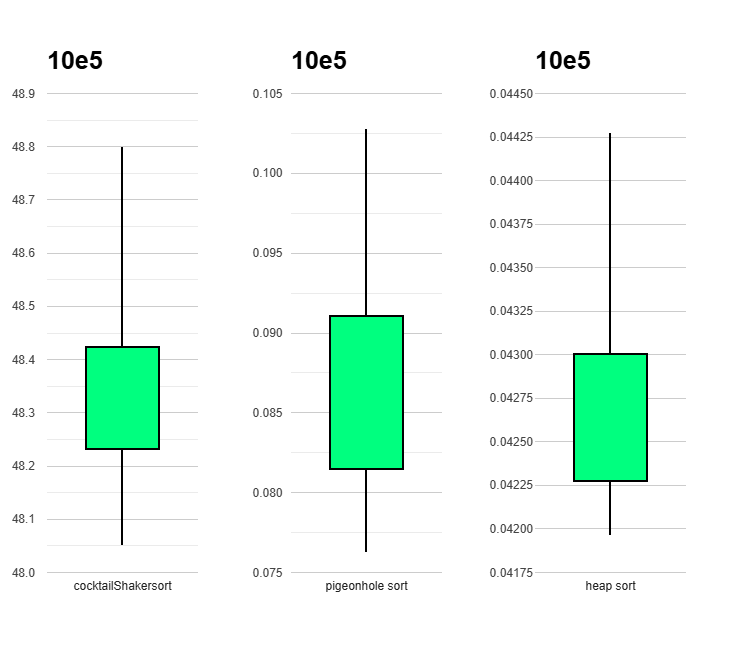


Рис. 5 График времени для длины массива 10е5

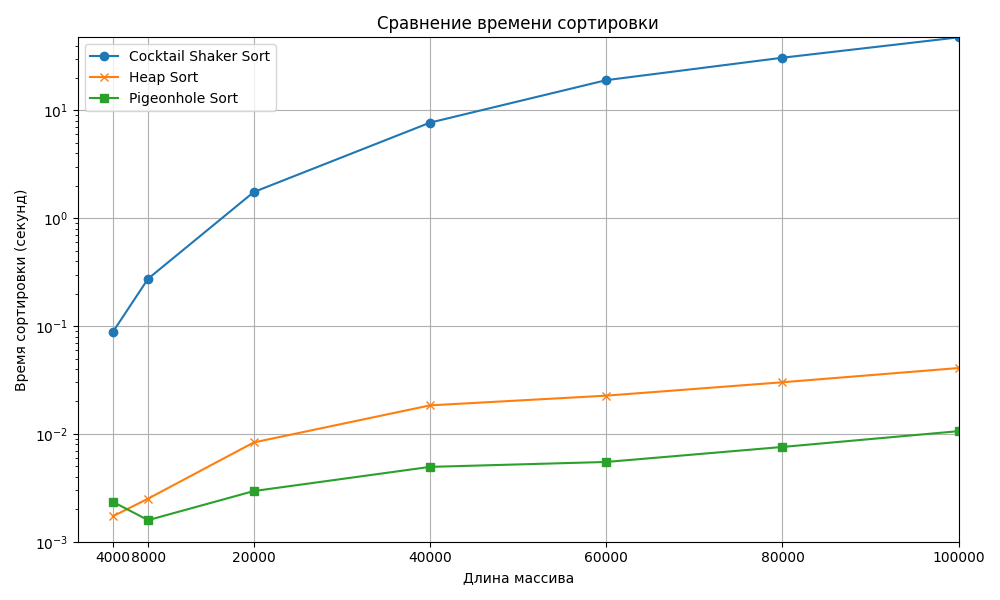


Рис. 6 Линейный график для небольшого времени

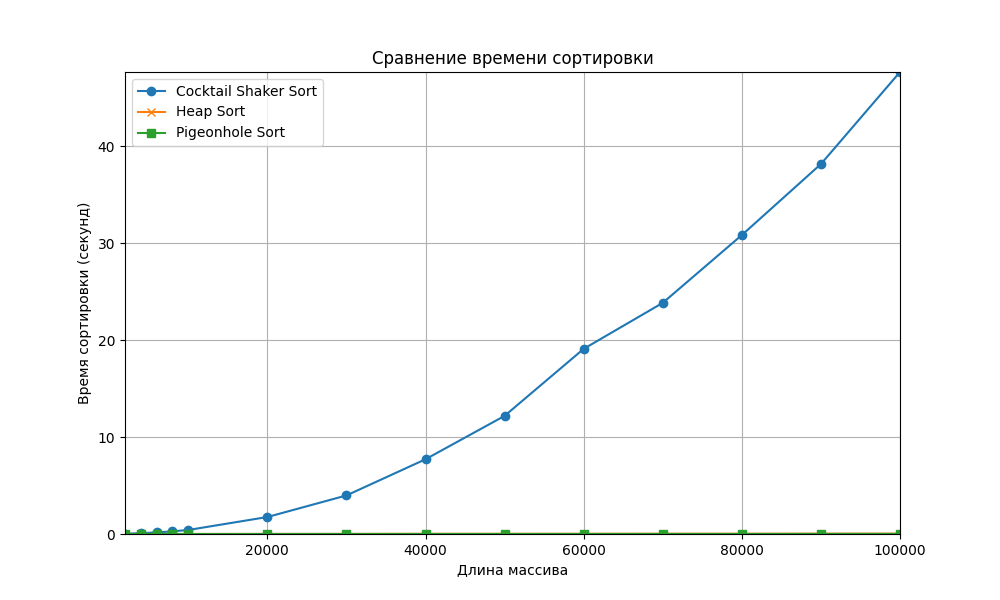


Рис. 7 Линейный график для большого времени

**Анализ графиков**

**График 1** визуализирует время работы трех алгоритмов сортировки (Cocktail Shaker Sort, Heap Sort и Pigeonhole Sort) для массивов размером до 10,000:

* **Cocktail Shaker Sort**:
  + График показывает резкое увеличение времени выполнения при увеличении размера массива.
  + Время выполнения стремительно возрастает, особенно после 5,000 элементов, что подтверждает худшую временную сложность O(n²).
  + Такой рост времени указывает на низкую эффективность алгоритма при больших входных массивах, что делает его непрактичным для реальных задач сортировки.
* **Heap Sort**:
  + На графике видно, что время выполнения увеличивается более линейно и остается в пределах O(n log n).
  + Этот алгоритм демонстрирует устойчивую и приемлемую производительность для массивов даже на больших значениях, показывая, что он лучше подходит для задач, где требуется сортировка больших объемов данных.
* **Pigeonhole Sort**:
  + В отличие от двух других алгоритмов, Pigeonhole Sort показывает относительно низкое время выполнения по сравнению с размером массива.
  + Он остается стабильным на уровне, так как данные эффективно распределяются по карманам.
  + Однако его эффективность зависит от диапазона значений, что также стоит учитывать при анализе его использования.

**График 2** отображает время работы тех же сортировок для массивов размером до 100,000:

* **Cocktail Shaker Sort**:
  + Здесь видно, что алгоритм показывает крайне негативные результаты. Время выполнения резко увеличивается с каждым добавленным элементом, крайне затрудняя обработку массивов.
  + Это еще раз подтверждает временную сложность O(n²), так как алгоритм требует гораздо больше времени для каждого прохода по массиву.
* **Heap Sort**:
  + Продолжает показывать стабильный рост времени выполнения, оставаясь в рамках O(n log n).
  + На данный момент время работы не возрастает так значительно, как у Cocktail Shaker Sort, что делает его предпочтительным выбором для больших данных.
* **Pigeonhole Sort**:
  + Эффективность также сохраняется, оставаясь ниже, чем у Cocktail Shaker Sort, но может достигать значений, сопоставимых с Heap Sort, в зависимости от распределения значений в данных.
  + Есть предположение, что с растущими диапазонами значений его производительность начинает снижаться.

**Анализ диаграмм 3-4**

**График 3** демонстрирует ящики с усами (box plot) для времени сортировки с 10е5 элементами:

* **Cocktail Shaker Sort**:
  + Показаны значительные вариации по времени выполнения.
  + Широкий ящик указывает на большие отклонения, особенно на верхнем конце, что говорит о том, что в некоторых случаях алгоритм работает значительно медленнее.
* **Pigeonhole Sort** и **Heap Sort**:
  + Обе имеют более узкие ящики, что указывает на стабильность производительности.
  + Их медленное время менее подвержено колебаниям, что подтверждает их больший потенциал для использования в задачах с большими объемами данных.

**График 4** аналогичен третьему, но показывает данные для массивов длиной 104:

* **Cocktail Shaker Sort**:
  + Ящик также довольно широкий, но относительно уже.
  + Это указывает на меньшие, но все еще значительные колебания во времени выполнения — алгоритм, хоть и работает быстрее, всё равно неэффективен.
* **Heap Sort** и **Pigeonhole Sort**:
  + Показывают схожие результаты, заметно меньше отклонений, что делает их более предсказуемыми и эффективными.
  + Время выполнения для обеих сортировок меньше и стабильнее, что подтверждает их производительность на практике.

1. Заключение

В ходе выполнения работы были реализованы три различных алгоритма сортировки: коктейльное встряхивание (Cocktail Shaker Sort), кучевая сортировка (Heap Sort) и сортировка по карманам (Pigeonhole Sort). Цель работы была достигнута путем тестирования каждой из сортировок на массивах различной длины, с анализом временных характеристик и производительности.

Разработка алгоритмов включала в себя анализ их теоретических особенностей, реализацию на языке C++, а также тщательное тестирование с использованием заранее подготовленных массивов. Полученные результаты были задокументированы, и все тесты подтвердили корректность работы алгоритмов.

В измерениях времени выполнения по каждому алгоритму было отмечено, что сортировка коктейльным встряхиванием показала наихудшие результаты на больших объемах данных, в то время как кучевая сортировка и сортировка по карманам продемонстрировали более стабильную производительность. В среднем и худшем случаях часоты выполнения алгоритмов соответствуют теоретическим оценкам, что свидетельствует о высокой эффективности выбранных методов сортировки.

В качестве перспективы для дальнейших исследований можно рассмотреть оптимизацию алгоритмов с точки зрения уменьшения затрат по памяти и времени. Также будет целесообразно протестировать их на различных подмножествах данных и в условиях параллельной обработки, что может значительно улучшить производительность на больших объемах информации.

1. Приложения

Листинг кода файла Lab5.cpp

#include <iostream>

#include <vector>

#include <algorithm>

#include <cassert>

#include <chrono>

#include <climits>

#include <fstream>

#include <string>

// Функция сортировки коктейльным встряхиванием (Cocktail Shaker Sort)

void cocktailShakerSort(std::vector<int>& arr) {

bool swapped = true; // O(1) - флаг для отслеживания обменов

int start = 0; // O(1) - начало рассматриваемого диапазона

size\_t end = arr.size() - 1; // O(1) - конец рассматриваемого диапазона

// Внешний цикл, выполняется, пока есть обмены

while (swapped) { // O(n) в лучшем случае, O(n^2) в худшем случае

swapped = false; // O(1) - сброс флага перед новым проходом

// Прямой проход: от начала к концу

for (int i = start; i < end; i++) { // O(n)

// Если текущий элемент больше следующего, меняем их местами

if (arr[i] > arr[i + 1]) { // O(1) - сравнение

std::swap(arr[i], arr[i + 1]); // O(1) - обмен элементов

swapped = true; // O(1) - установление флага, что был обмен

}

}

if (!swapped) break; // O(1) - проверка флага

end--; // O(1) - уменьшаем конец диапазона

swapped = false; // O(1) - сброс флага для обратного прохода

// Обратный проход: от конца к началу

for (int i = end; i > start; --i) { // O(n)

// Если текущий элемент меньше предыдущего, меняем их местами

if (arr[i] < arr[i - 1]) { // O(1) - сравнение

std::swap(arr[i], arr[i - 1]); // O(1) - обмен элементов

swapped = true; // O(1) - установление флага, что был обмен

}

}

start++; // O(1) - увеличиваем начало диапазона

}

}

/\*

Временная сложность:

- В худшем случае (обратный порядок): O(n^2), так как каждый элемент может быть перемещен

во время каждого прохода.

- В лучшем случае (уже отсортированный массив): O(n) за один проход без обменов.

- В среднем случае: O(n^2), учитывая, что массив случайно перемешан.

Пространственная сложность: O(1), используется только несколько переменных для хранения

текущего состояния и флагов.

\*/

// Функция для перестройки поддерева в кучу (heap)

void heapify(std::vector<int>& arr, int n, int i) {

int largest = i; // O(1) — инициализация корня как самого большого элемента

int left = 2 \* i + 1; // O(1) — индекс левого дочернего элемента

int right = 2 \* i + 2; // O(1) — индекс правого дочернего элемента

// Проверяем, если левый дочерний элемент больше корня

if (left < n && arr[left] > arr[largest]) { // O(1)

largest = left; // O(1)

}

// Проверяем, если правый дочерний элемент больше текущего самого большого элемента

if (right < n && arr[right] > arr[largest]) { // O(1)

largest = right; // O(1)

}

// Если самый большой элемент не корень, меняем местами

if (largest != i) { // O(1)

std::swap(arr[i], arr[largest]); // O(1) - обмен значениями

heapify(arr, n, largest); // O(log n) - рекурсивный вызов

}

}

/\*

Временная сложность heapify:

- O(log n) в среднем и худшем случае, так как может быть выполнено до log(n) рекурсивных вызовов

- O(1) в лучшем случае, если элементы уже расположены правильно (например, когда самый большой элемент является корнем)

Пространственная сложность: O(1) — используется лишь несколько переменных для хранения состояния.

\*/

void heapSort(std::vector<int>& arr) {

size\_t n = arr.size(); // O(1) — получаем размер массива

// Построение кучи (перестройка массива в max-heap)

for (int i = n / 2 - 1; i >= 0; i--) { // O(n)

heapify(arr, n, i); // O(log n)

}

// Один за другим извлекаем элементы из кучи

for (int i = n - 1; i >= 0; i--) { // O(n)

// Перемещаем текущий корень (максимальный элемент) в конец

std::swap(arr[0], arr[i]); // O(1)

heapify(arr, i, 0); // O(log n)

}

}

/\*

Временная сложность heapSort:

- O(n log n) в худшем и среднем случае, так как построение кучи (n/2 вызовов heapify)

занимает O(n) и каждая перестройка биов составляет O(log n).

- O(n log n) в лучшем случае (уже отсортирована), так как нужно пройти все элементы.

Пространственная сложность: O(1) — не используются дополнительные массивы.

\*/

// Функция для сортировки по карманам (Pigeonhole Sort)

void pigeonholeSort(std::vector<int>& arr) {

size\_t n = arr.size(); // O(1) — получаем размер массива

if (n <= 0) return; // O(1) — если массив пуст, выходим

// Находим минимальное и максимальное значения в массиве

int min\_value = \*std::min\_element(arr.begin(), arr.end()); // O(n) — нахождение минимального элемента

int max\_value = \*std::max\_element(arr.begin(), arr.end()); // O(n) — нахождение максимального элемента

int range = max\_value - min\_value + 1; // O(1) — диапазон значений

std::vector<std::vector<int>> pigeonholes(range); // O(range) — создание вектора карманов

// Размещаем элементы в соответствующие карманы

for (int i = 0; i < n; i++) { // O(n)

pigeonholes[arr[i] - min\_value].push\_back(arr[i]); // O(1)

}

// Собираем элементы обратно в исходный массив

int index = 0; // O(1)

for (int i = 0; i < range; i++) { // O(range)

for (int j = 0; j < pigeonholes[i].size(); j++) { // O(k), где k — количество элементов в кармане

arr[index] = pigeonholes[i][j]; // O(1)

index++; // O(1)

}

}

}

/\*

Временная сложность:

- O(n) для поиска минимального и максимального элемента.

- O(n + range) для размещения элементов в карманы и сбора их обратно.

- В общем случае: O(n + range), где n — количество элементов, а range — диапазон значений.

- Если элементы имеют большой диапазон, может потребоваться много памяти, но незначительное количество операций.

Пространственная сложность: O(range), поскольку мы создаем массив карманов, который зависит от диапазона возможных значений.

\*/

void printArray(const std::vector<int>& arr) {

for (int num : arr) { // O(n)

std::cout << num << " "; // O(1)

}

std::cout << std::endl; // O(1)

}

// Универсальная функция для тестирования любой сортировки

void runTests(void (\*sortFunction)(std::vector<int>&)) {

// Тест: лучший случай

{

std::vector<int> arr = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

std::vector<int> expected = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }; // O(n)

std::cout << "Исходный массив (лучший случай): ";

printArray(arr);

sortFunction(arr);

assert(arr == expected); // O(n)

std::cout << "Отсортированный массив: ";

printArray(arr);

std::cout << "Тест 1 (Лучший случай) прошел." << std::endl << std::endl;

}

// Тест: средний случай

{

std::vector<int> arr = { 8, 3, 5, 2, 7, 1, 4, 6 };

std::vector<int> expected = { 1, 2, 3, 4, 5, 6, 7, 8 }; // O(n)

std::cout << "Исходный массив (средний случай): ";

printArray(arr);

sortFunction(arr);

assert(arr == expected); // O(n)

std::cout << "Отсортированный массив: ";

printArray(arr);

std::cout << "Тест 2 (Средний случай) прошел." << std::endl << std::endl;

}

// Тест: худший случай

{

std::vector<int> arr = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };

std::vector<int> expected = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }; // O(n)

std::cout << "Исходный массив (худший случай): ";

printArray(arr);

sortFunction(arr);

assert(arr == expected); // O(n)

std::cout << "Отсортированный массив: ";

printArray(arr);

std::cout << "Тест 3 (Худший случай) прошел." << std::endl << std::endl;

}

std::cout << "Все тесты пройдены!" << std::endl;

}

// Структура для хранения результатов сортировки

struct SortResult {

std::string algorithm; // Название алгоритма

size\_t arraySize; // Размер массива

double elapsedTime; // Время сортировки

};

//// Функция для чтения данных из файла

//std::vector<int> readFile(const std::string& filename) {

// std::ifstream file(filename);

// std::vector<int> data;

// int value;

//

// // Проверяем, открылся ли файл

// if (!file.is\_open()) {

// std::cerr << "Ошибка при открытии файла: " << filename << std::endl;

// return data; // Возвращаем пустой вектор в случае ошибки

// }

//

// // Чтение значений из файла

// while (file >> value) {

// data.push\_back(value);

// }

//

// file.close(); // Закрываем файл

// return data; // Возвращаем вектор данных

//}

//

//// Тестирование времени для всех трех сортировок

//void Test\_time(const std::vector<std::string>& filenames,

// std::vector<double>& timeCocktail,

// std::vector<double>& timeHeap,

// std::vector<double>& timePigeonhole) {

// for (const std::string& filename : filenames) {

// // Читаем данные из файла

// std::vector<int> data = readFile(filename);

//

// if (data.empty()) continue; // Пропустить пустой вектор

//

// // Измеряем время сортировки коктейльным встряхиванием

// auto start = std::chrono::high\_resolution\_clock::now();

// cocktailShakerSort(data);

// auto end = std::chrono::high\_resolution\_clock::now();

// std::chrono::duration<double> elapsed = end - start;

// timeCocktail.push\_back(elapsed.count());

//

// // Читаем данные снова для следующей сортировки

// data = readFile(filename);

//

// // Измеряем время сортировки кучей

// start = std::chrono::high\_resolution\_clock::now();

// heapSort(data);

// end = std::chrono::high\_resolution\_clock::now();

// elapsed = end - start;

// timeHeap.push\_back(elapsed.count());

//

// // Читаем данные снова для следующей сортировки

// data = readFile(filename);

//

// // Измеряем время сортировки по карманам

// start = std::chrono::high\_resolution\_clock::now();

// pigeonholeSort(data);

// end = std::chrono::high\_resolution\_clock::now();

// elapsed = end - start;

// timePigeonhole.push\_back(elapsed.count());

// }

//}

//

//// Функция для записи времени в файл

//void writeTimesToFile(const std::string& filename,

// const std::vector<double>& times) {

// std::ofstream outFile(filename);

// if (!outFile.is\_open()) {

// std::cerr << "Ошибка при открытии файла для записи: " << filename << std::endl;

// return;

// }

//

// // Запись каждого времени на новой строке

// for (const double& time : times) {

// outFile << time << std::endl;

// }

//

// outFile.close(); // Закрываем файл

//}

//

//int main() {

// setlocale(LC\_ALL, "Ru");

//

// // Файлы для тестирования на массиве длиной 10^4

// std::cout << "Тестирование на массиве длиной 10^4:" << std::endl;

// std::vector<std::string> box\_plot\_files\_10e4;

// for (int i = 0; i < 50; i++) {

// box\_plot\_files\_10e4.push\_back("file\_boxplot\_10e4/file\_" + std::to\_string(i) + ".txt");

// }

//

// // Векторы для хранения времени выполнения сортировок

// std::vector<double> timeCocktail\_10e4;

// std::vector<double> timeHeap\_10e4;

// std::vector<double> timePigeonhole\_10e4;

//

// // Запускаем тесты

// Test\_time(box\_plot\_files\_10e4, timeCocktail\_10e4, timeHeap\_10e4, timePigeonhole\_10e4);

//

// // Записываем времена в файл

// writeTimesToFile("times\_10e4.txt", timeCocktail\_10e4);

// writeTimesToFile("times\_10e4\_heap.txt", timeHeap\_10e4);

// writeTimesToFile("times\_10e4\_pigeonhole.txt", timePigeonhole\_10e4);

//

// // Файлы для тестирования на массиве длиной 10^5

// std::cout << "Тестирование на массиве длиной 10^5:" << std::endl;

// std::vector<std::string> box\_plot\_files\_10e5;

// for (int i = 0; i < 50; i++) {

// box\_plot\_files\_10e5.push\_back("file\_boxplot\_10e5/file\_" + std::to\_string(i) + ".txt");

// }

//

// // Векторы для хранения времени выполнения сортировок

// std::vector<double> timeCocktail\_10e5;

// std::vector<double> timeHeap\_10e5;

// std::vector<double> timePigeonhole\_10e5;

//

// // Запускаем тесты

// Test\_time(box\_plot\_files\_10e5, timeCocktail\_10e5, timeHeap\_10e5, timePigeonhole\_10e5);

//

// // Записываем времена в файл

// writeTimesToFile("times\_10e5.txt", timeCocktail\_10e5);

// writeTimesToFile("times\_10e5\_heap.txt", timeHeap\_10e5);

// writeTimesToFile("times\_10e5\_pigeonhole.txt", timePigeonhole\_10e5);

//

// return 0;

//}

//// Функция для чтения данных из файла

//std::vector<int> readFile(const std::string& filename) {

// std::ifstream file(filename);

// std::vector<int> data;

// int value;

//

// // Проверяем, открылся ли файл

// if (!file.is\_open()) {

// std::cerr << "Ошибка при открытии файла: " << filename << std::endl;

// return data; // Возвращаем пустой вектор в случае ошибки

// }

//

// // Чтение значений из файла

// while (file >> value) {

// data.push\_back(value);

// }

//

// file.close(); // Закрываем файл

// return data; // Возвращаем вектор данных

//}

//

//// Функция для записи результатов в файл

//void writeResultsToFile(const std::string& filename, const std::vector<std::string>& results) {

// std::ofstream outFile(filename);

// if (!outFile.is\_open()) {

// std::cerr << "Ошибка при открытии файла для записи: " << filename << std::endl;

// return;

// }

//

// // Запись результатов в файл

// for (const auto& result : results) {

// outFile << result << std::endl; // Каждое значение на новой строке

// }

//

// outFile.close(); // Закрываем файл

//}

//

//// Тестирование времени для всех трех сортировок

//void Test\_time(const std::vector<std::string>& filenames,

// std::vector<std::string>& resultsCocktail,

// std::vector<std::string>& resultsHeap,

// std::vector<std::string>& resultsPigeonhole) {

// for (const std::string& filename : filenames) {

// // Читаем данные из файла

// std::vector<int> data = readFile(filename);

//

// if (data.empty()) continue; // Пропустить пустой вектор

//

// // Измеряем время сортировки коктейльным встряхиванием

// auto start = std::chrono::high\_resolution\_clock::now();

// cocktailShakerSort(data);

// auto end = std::chrono::high\_resolution\_clock::now();

// std::chrono::duration<double> elapsed = end - start;

// resultsCocktail.push\_back(std::to\_string(data.size()) + " - " + std::to\_string(elapsed.count()));

//

// // Читаем данные снова для следующей сортировки

// data = readFile(filename);

//

// // Измеряем время сортировки кучей

// start = std::chrono::high\_resolution\_clock::now();

// heapSort(data);

// end = std::chrono::high\_resolution\_clock::now();

// elapsed = end - start;

// resultsHeap.push\_back(std::to\_string(data.size()) + " - " + std::to\_string(elapsed.count()));

//

// // Читаем данные снова для следующей сортировки

// data = readFile(filename);

//

// // Измеряем время сортировки по карманам

// start = std::chrono::high\_resolution\_clock::now();

// pigeonholeSort(data);

// end = std::chrono::high\_resolution\_clock::now();

// elapsed = end - start;

// resultsPigeonhole.push\_back(std::to\_string(data.size()) + " - " + std::to\_string(elapsed.count()));

// }

//}

//

//int main() {

// setlocale(LC\_ALL, "Ru");

//

// // Файлы для тестирования на массиве длиной 10^4

// std::cout << "Тестирование на массиве длиной 10^4:" << std::endl;

// std::vector<std::string> box\_plot\_files\_10e4;

// for (int i = 0; i < 50; i++) {

// box\_plot\_files\_10e4.push\_back("test\_input\_1000\_to\_10000/file\_" + std::to\_string(i) + ".txt");

// }

//

// // Векторы для хранения результатов

// std::vector<std::string> resultsCocktail\_10e4;

// std::vector<std::string> resultsHeap\_10e4;

// std::vector<std::string> resultsPigeonhole\_10e4;

//

// // Запускаем тесты

// Test\_time(box\_plot\_files\_10e4, resultsCocktail\_10e4, resultsHeap\_10e4, resultsPigeonhole\_10e4);

//

// // Записываем результаты в файлы

// writeResultsToFile("results\_cocktail\_10e4.txt", resultsCocktail\_10e4);

// writeResultsToFile("results\_heap\_10e4.txt", resultsHeap\_10e4);

// writeResultsToFile("results\_pigeonhole\_10e4.txt", resultsPigeonhole\_10e4);

//

// // Файлы для тестирования на массиве длиной 10^5

// std::cout << "Тестирование на массиве длиной 10^5:" << std::endl;

// std::vector<std::string> box\_plot\_files\_10e5;

// for (int i = 0; i < 50; i++) {

// box\_plot\_files\_10e5.push\_back("test\_input\_1e5/file\_" + std::to\_string(i) + ".txt");

// }

//

// // Векторы для хранения результатов

// std::vector<std::string> resultsCocktail\_10e5;

// std::vector<std::string> resultsHeap\_10e5;

// std::vector<std::string> resultsPigeonhole\_10e5;

//

// // Запускаем тесты

// Test\_time(box\_plot\_files\_10e5, resultsCocktail\_10e5, resultsHeap\_10e5, resultsPigeonhole\_10e5);

//

// // Записываем результаты в файлы

// writeResultsToFile("results\_cocktail\_10e5.txt", resultsCocktail\_10e5);

// writeResultsToFile("results\_heap\_10e5.txt", resultsHeap\_10e5);

// writeResultsToFile("results\_pigeonhole\_10e5.txt", resultsPigeonhole\_10e5);

//

// return 0;

//}

int main() {

setlocale(LC\_ALL, "Ru");

std::cout << "Тестирование сортировки коктейльным встряхиванием:" << std::endl;

runTests(cocktailShakerSort);

std::cout << "\nТестирование сортировки кучей (Heap Sort):" << std::endl;

runTests(heapSort);

std::cout << "\nТестирование сортировки по карманам (Pigeonhole Sort):" << std::endl;

runTests(pigeonholeSort);

return 0;

}