ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 5

«Bubble sort, Heap sort, Bucket sort»

Выполнил работу

Крапивина Яна

Академическая группа №J3112

Принято

Должность, звание Фамилия Имя преподавателя

Максим Дунаев, практик

Санкт-Петербург

2024

**Структура отчёта:**

1. Введение

Цель работы: проанализировать выбранные алгоритмы.

Задачи:

* Посчитать память;
* Посчитать асимптотику;
* Проверить код на работоспособность;
* Реализовать графики работы алгоритмов

1. Теоретическая подготовка

* Bubble sort

Используемые типы данных:

* Int: используется для хранения элементов массива и индексов;
* Bool: для отслеживания, произошла ли перестановка элементов в текущей итерации.

Используемый алгоритм:

Сортировка пузырьком

1. Внешний цикл: Проходит по массиву от первого до предпоследнего элемента.

2. Внутренний цикл: Проходит по неотсортированной части массива, сравнивая соседние элементы.

3. Сравнение и обмен: Если два соседних элемента находятся не в правильном порядке (например, если первый больше второго при сортировке по возрастанию), они меняются местами.

4. Проверка на завершение: Если в ходе прохода внутреннего цикла не произошло ни одной перестановки, это означает, что массив уже отсортирован, и внешний цикл прерывается.

* Heap sort

Используемые типы данных: int(для целых чисел, считанных из файла)

Алгоритм сортировки:

* Построение кучи;
* Извлечение элементов из кучи пока её размер не станет равен нулю.
* Bucket sort

Используемые типы данных: int(базовый тип данных, используемый для хранения целых чисел).

Алгоритм:

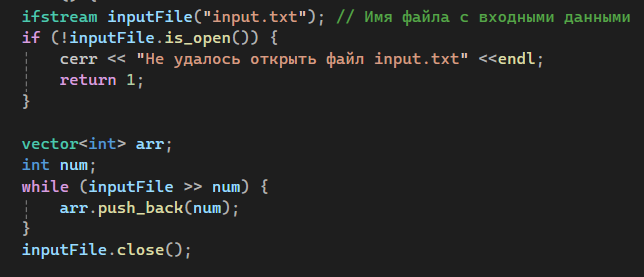
• sort(buckets[i].begin(), buckets[i].end()): Стандартный алгоритм сортировки из библиотеки <algorithm>. Используется для сортировки элементов внутри каждого ведра.

• swap(): Функция обмена значениями двух переменных, используемая внутри sort().

1. Реализация

Процесс работы Bubble sort:

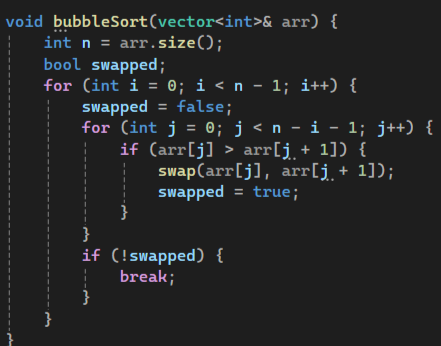
Чтение данных из файла



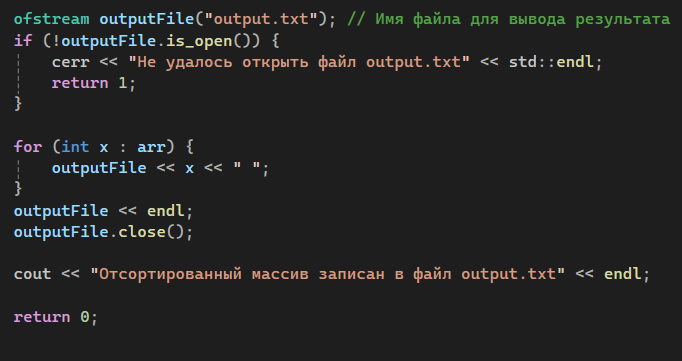
Сортировка массива:

bubbleSort(arr);

Функция bubbleSort:



Запись отсортированного массива в файл:



Используемые библиотеки:

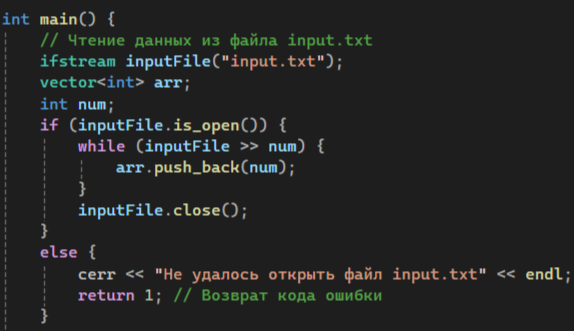
• <iostream>: для ввода/вывода данных в консоль.

• <fstream>: для работы с файлами.

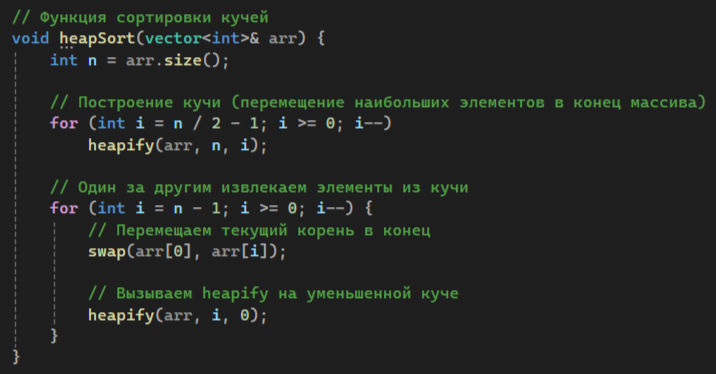
• <vector>: для хранения массива чисел.

Процесс работы Heap sort:

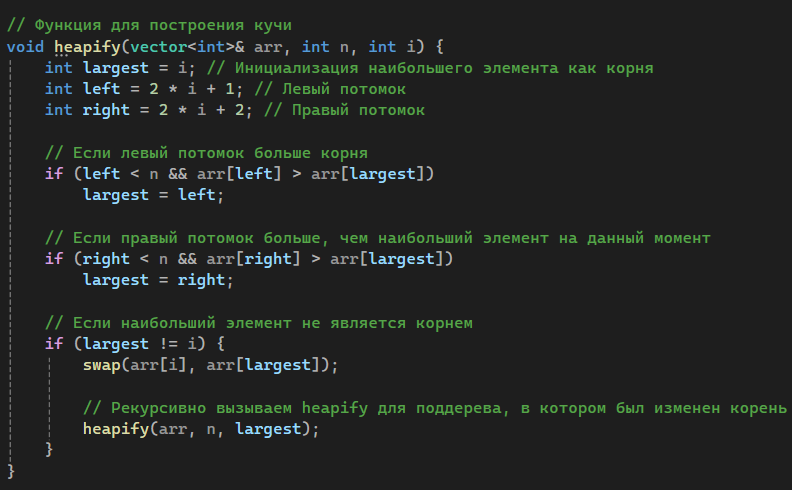
Чтение данных из файла:



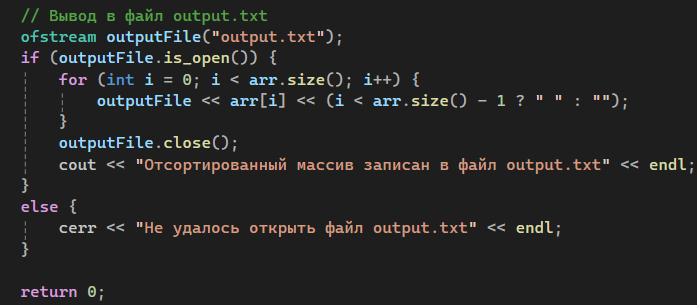
Сортировка массива:



Вызов функции heapify внутри сортировки:



Запись отсортированного массива в файл:

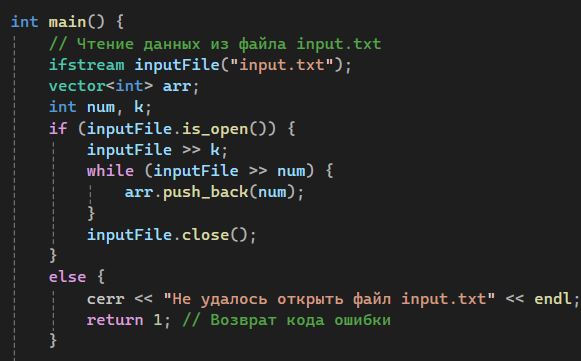


Используемые библиотеки:

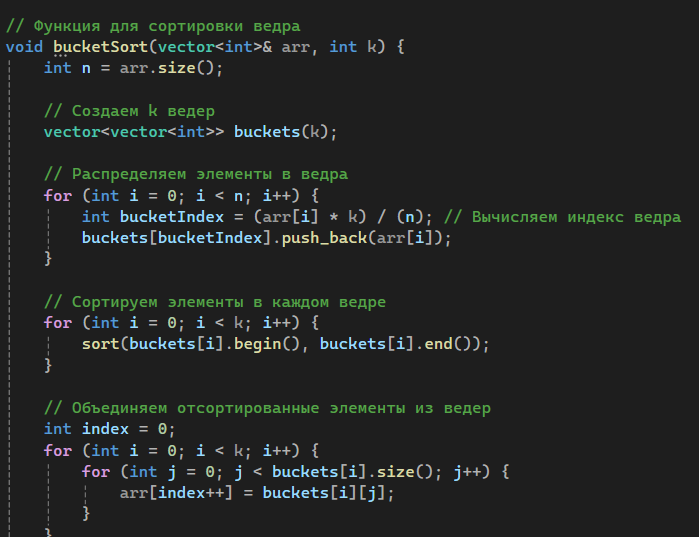
1. <iostream>: для вывода сообщения о завершении работы программы и сообщения об ошибках.
2. <vector>: используется для хранения чисел, считанных из входного файла.
3. <algorithm>: используется функция swap() для обмена местами элементов массива.
4. <fstream>: предоставляет классы для работы с файлами.

Процесс работы Bucket sort:

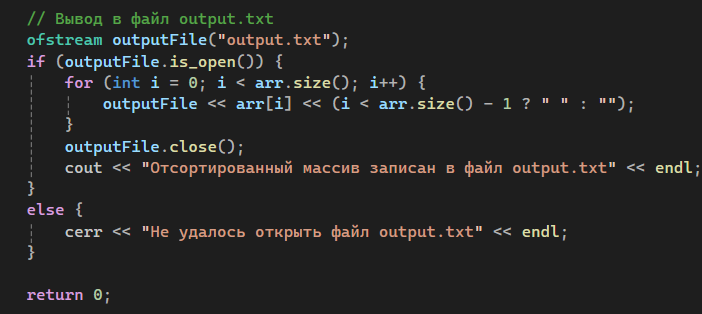
Чтение данных из файла:



Сортировка массива:



Запись отсортированного массива в файл:



1. Экспериментальная часть

Подсчет памяти и асимптотики:

Bubble sort:

* память: 4\*n + C
* время: в худшем и среднем случаях составляет O(n^2), в лучшем случае(массив уже отсортирован) составляет O(n).

Heap sort:

* память: сортирует элементы исходного массива, не создавая дополнительных массивов для хранения промежуточных результатов + используются переменные, не зависящие от входного массива: О(1)
* время: O(n) (или O(n log n) в худшем случае) для построения кучи.

O(n log n) для извлечения элементов из кучи.

Общая асимптотическая сложность HeapSort составляет O(n log n).

Bucket sort:

* память: О(n)
* время: в худшем случае: O(n log n) (например, если все элементы попадают в одно ведро), в среднем случае: O(n) (если элементы распределены равномерно по ведрам).

В этом разделе вам необходимо привести результаты работы вашего алгоритма, с таблицами и графиками, демонстрирующими выполнения алгоритма с различными условиями и наборами данных. Оценивается производительность и сравниваются результаты с теоретическими оценками.

Подсчёт по памяти (только для циклов и сложных структур) – как в лабораторной работе №2.

Подсчёт асимптотики (только для циклов и сложных структур) – как в лабораторной работе №3.

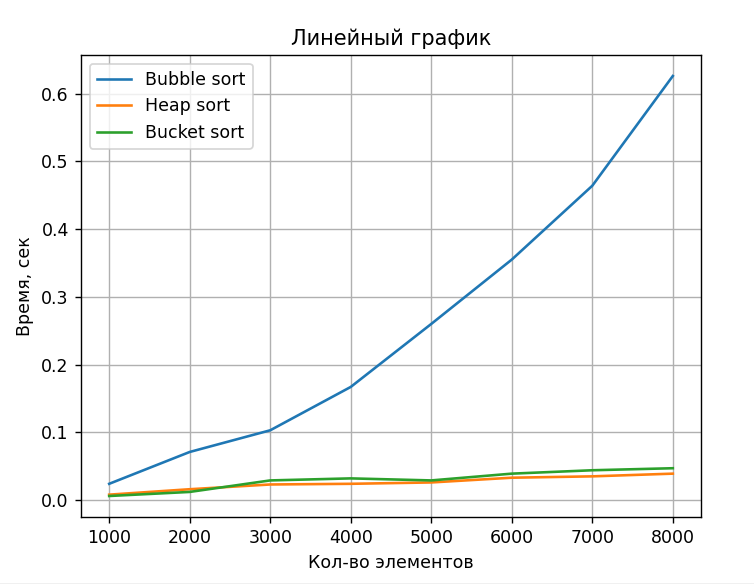
График зависимости времени от числа элементов. Пример выполнения:

Согласно требованиям моего варианта, на вход к моему алгоритму подаётся до 25 элементов. Теоретически заданная сложность задачи составляет O(2^N) и более. Для тестирования алгоритма была собрана статистика, приведенная в таблице №\*.

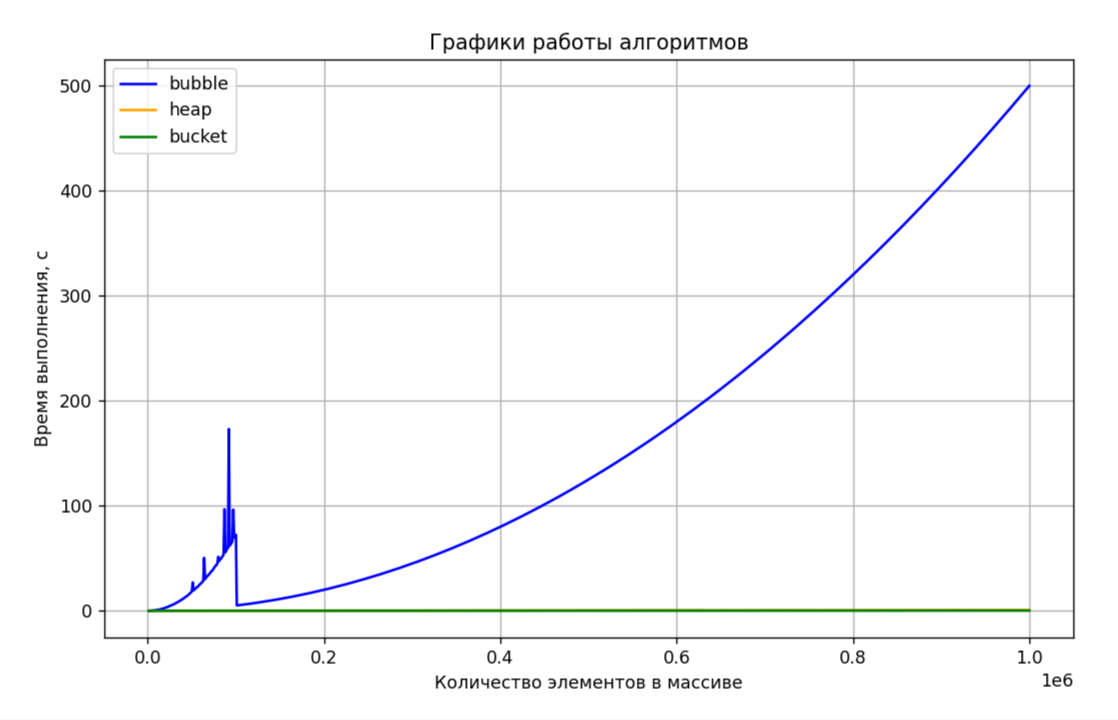
Таблица №1 - Подсчёт сложности реализованных алгоритмов

|  |  |  |  |
| --- | --- | --- | --- |
| Размер | Bubble sort | Heap sort | Bucket sort |
| 1000 | 0,024 | 0,008 | 0,006 |
| 2000 | 0,071 | 0,016 | 0,012 |
| 3000 | 0,103 | 0,023 | 0,029 |
| 4000 | 0,167 | 0,024 | 0,032 |
| 5000 | 0,260 | 0,026 | 0,029 |
| 6000 | 0,355 | 0,033 | 0,039 |
| 7000 | 0,464 | 0,035 | 0,044 |
| 8000 | 0,626 | 0,039 | 0,047 |

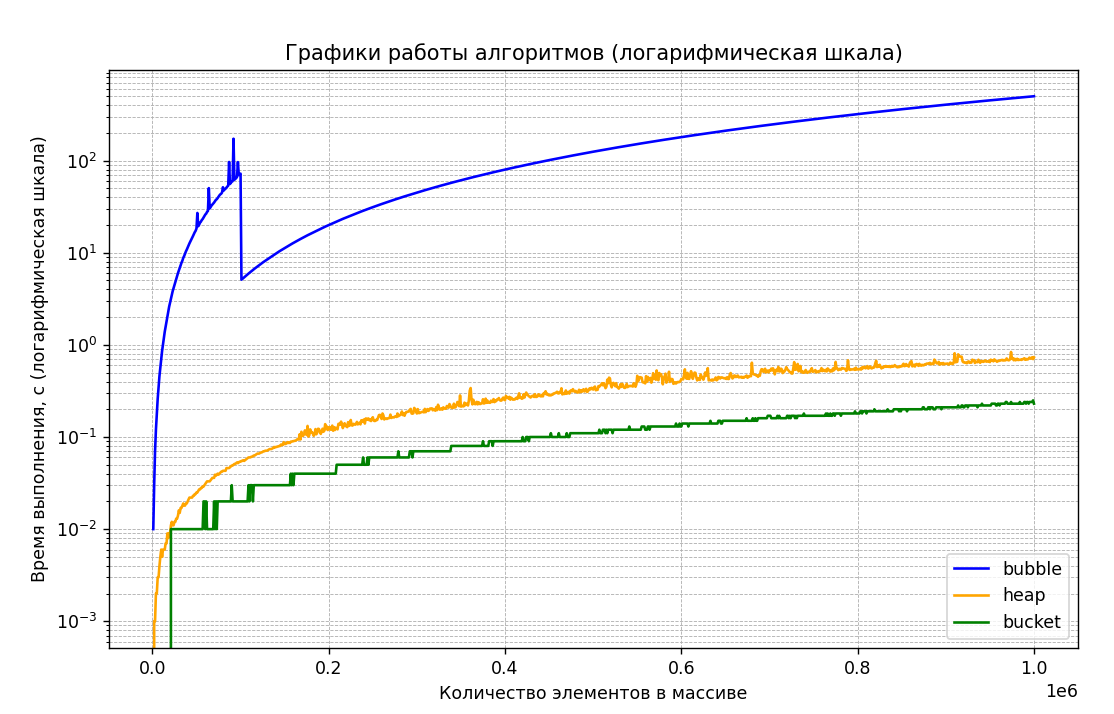
График представляющий визуально удобный формат данных из таблицы №1 представлен на изображении №1.



Изображение №1 – линейный график до 8000 элементов

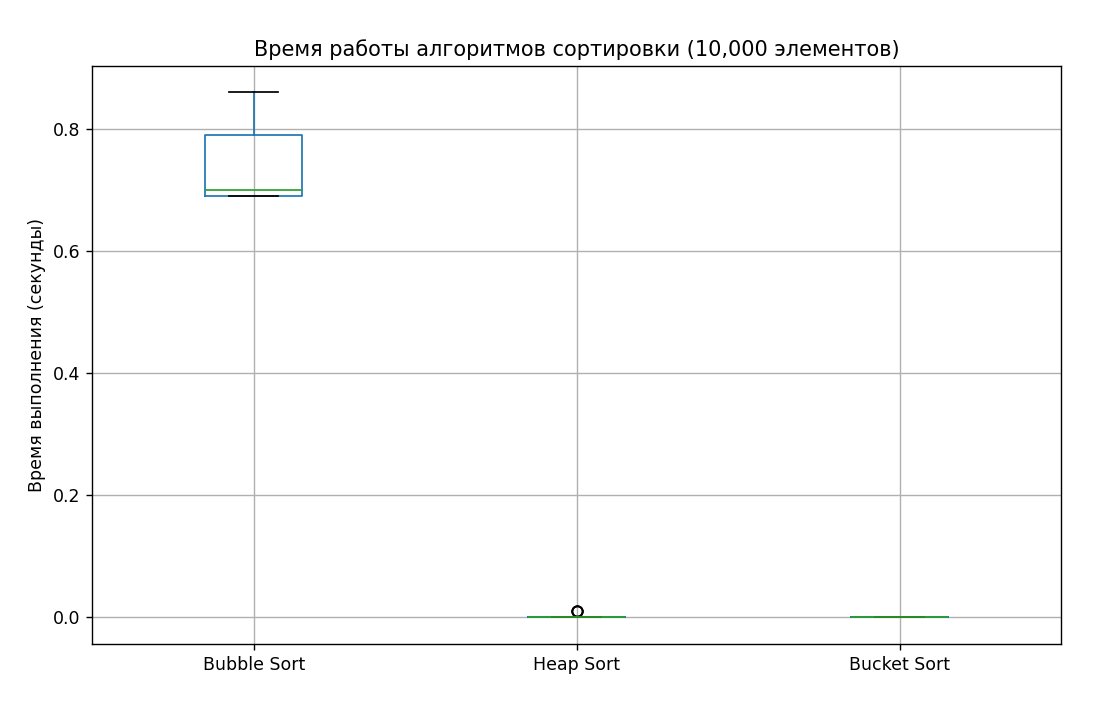


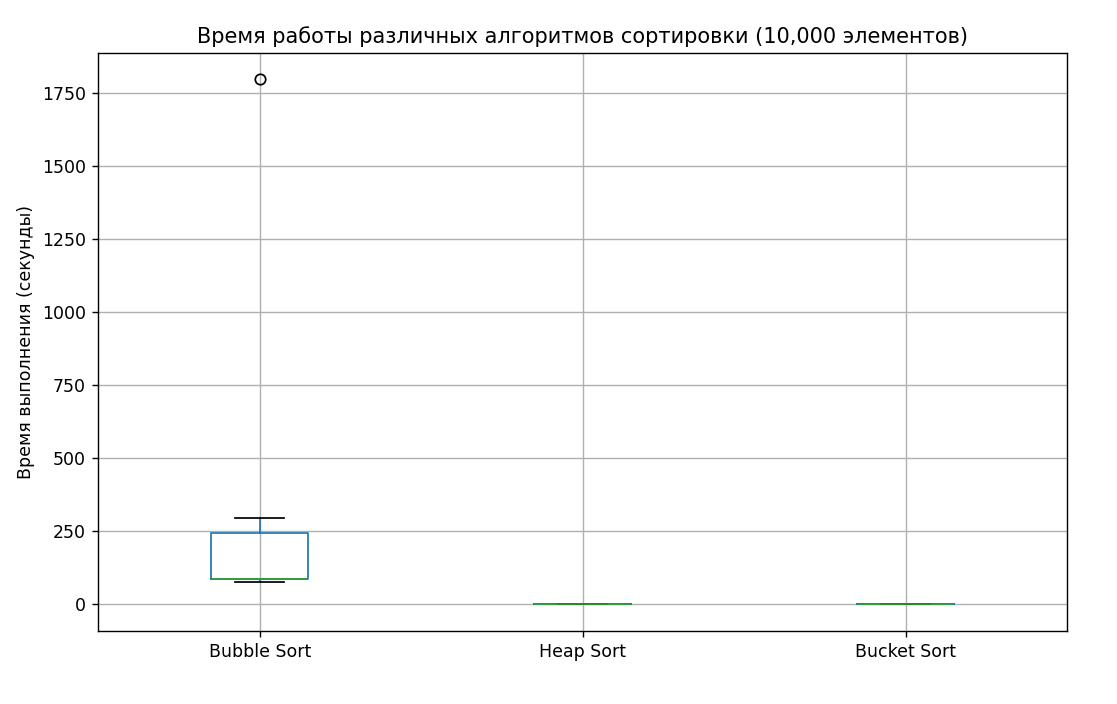
Изображение №2 – линейный график до 1е6 элементов



Изображение №3 – логарифмический график для наглядности

|  |  |  |  |
| --- | --- | --- | --- |
| Размер | Bubble sort | Heap sort | Bucket sort |
| 500 | 0,006 | 0,014 | 0,003 |
| 1000 | 0,024 | 0,008 | 0,006 |
| 1500 | 0,028 | 0,027 | 0,018 |
| 2000 | 0,071 | 0,016 | 0,012 |
| 2500 | 0,064 | 0,023 | 0,028 |
| 3000 | 0,103 | 0,023 | 0,029 |
| 3500 | 0,12 | 0,03 | 0,028 |
| 4000 | 0,167 | 0,024 | 0,032 |
| 4500 | 0,18 | 0,034 | 0,036 |
| 5000 | 0,260 | 0,026 | 0,029 |
| 5500 | 0,259 | 0,037 | 0,044 |
| 6000 | 0,355 | 0,033 | 0,039 |
| 6500 | 0,364 | 0,046 | 0,047 |
| 7000 | 0,464 | 0,035 | 0,044 |
| 7500 | 0,483 | 0,048 | 0,046 |
| 8000 | 0,626 | 0,039 | 0,047 |
| 8500 | 0,594 | 0,052 | 0,05 |
| 9000 | 0,658 | 0,055 | 0,053 |
| 9500 | 0,722 | 0,056 | 0,056 |
| 10000 | 0,794 | 0,063 | 0,058 |





Анализ графика и таблицы:

* Как мы видим, при больших значениях массивов 1 алгоритм работает намного медленнее, чем 2 и 3. При этом 2 работает дольше 3.
* Стабильность 1 алгоритма мала по сравнению с алгоритмами 2 и 3

1. Заключение

В ходе выполнения работы мною были реализованы алгоритмы сортировки. Цель работы была достигнута путём тестирования на числовых массивах с различным количеством элементов. Полученные результаты также совпадают с теоретическими оценками сложности алгоритма.

В ходе лабораторной работы, было выявлено, что асимптотика вполне соотносится с практикой, особенно на больших значениях. Так асимптотика 1 алгоритма много больше, чем асимптотика двух других, а также асимптотика 2 алгоритма больше 3, что можно увидеть на больших массивах.

При работе с массивами больших размеров наблюдаются сильные выбросы у 1 алгоритма, небольшие у 2, и почти отсутствуют выбросы у 3 алгоритма. Это зависит от устойчивости сортировки.

Где применимы алгоритмы:

Bubble sort

* Он часто используется в учебных целях для демонстрации базовых принципов алгоритмов сортировки.
* Из-за квадратичной временной сложности и большого количества операций, Bubble Sort не подходит для сортировки больших массивов данных.

Heap sort

* Heap Sort имеет временную сложность O(n log n) как в худшем, так и в среднем случае, что делает его очень эффективным для больших наборов данных.
* Heap Sort сортирует данные без использования дополнительной памяти, за исключением небольшого пространства для вспомогательных переменных. Это делает его подходящим для систем с ограниченными ресурсами памяти.

Bucket sort

* Bucket Sort – это эффективный алгоритм сортировки, который хорошо подходит для равномерно распределенных данных, где известен диапазон значений и требуется быстрая сортировка. Однако, в других случаях, другие алгоритмы сортировки могут быть более подходящими.

1. Приложения

ПРИЛОЖЕНИЕ А

Листинг кода файла Bubble sort.cpp

#include <iostream>

#include <vector>

#include <chrono>

#include <cassert>

using namespace std;

using namespace std::chrono;

void bubbleSort(vector<int>& arr) {

int n = arr.size();

bool swapped;

for (int i = 0; i < n - 1; i++) {

swapped = false;

for (int j = 0; j < n - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

swap(arr[j], arr[j + 1]);

swapped = true;

}

}

if (!swapped) {

break;

}

}

}

// Функция для печати массива

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

for (int num : arr) {

cout << num << " ";

}

cout << endl;

}

// Функция для проверки корректности сортировки с выводом результатов

void testBubbleSort() {

setlocale(LC\_ALL, "RU");

// Тестовые случаи

vector<pair<string, vector<int>>> testCases = {

{"Пустой массив", {}},

{"Отсортированный массив", {1, 2, 3, 4, 5}},

{"Неотсортированный массив", {5, 1, 4, 2, 8}},

{"Массив с повторяющимися элементами", {3, 1, 4, 1, 5, 9, 2, 6, 5}}

};

// Ожидаемые результаты

vector<vector<int>> expectedResults = {

{},

{1, 2, 3, 4, 5},

{1, 2, 4, 5, 8},

{1, 1, 2, 3, 4, 5, 5, 6, 9}

};

// Проход по каждому тестовому случаю

for (size\_t i = 0; i < testCases.size(); ++i) {

cout << "Тест " << i + 1 << ": " << testCases[i].first << endl;

vector<int> arr = testCases[i].second;

// Замер времени

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

bubbleSort(arr);

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

auto duration = duration\_cast<microseconds>(end - start);

// Проверка корректности

assert(arr == expectedResults[i]);

cout << "Исходный массив: ";

printArray(testCases[i].second);

cout << "Отсортированный массив: ";

printArray(arr);

cout << "Время выполнения: " << duration.count() << " микросекунд" << endl;

cout << "Тест успешно пройден!\n" << endl;

}

}

int main() {

// Вызов функции для тестирования

testBubbleSort();

return 0;

}

ПРИЛОЖЕНИЕ Б

Листинг кода файла Heap sort.cpp

#include <iostream>

#include <vector>

#include <algorithm>

#include <chrono>

#include <iomanip>

#include <cassert>

using namespace std;

using namespace std::chrono;

// Функция для построения кучи

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

int largest = i; // Инициализация наибольшего элемента как корня

int left = 2 \* i + 1; // Левый потомок

int right = 2 \* i + 2; // Правый потомок

// Если левый потомок больше корня

if (left < n && arr[left] > arr[largest])

largest = left;

// Если правый потомок больше, чем наибольший элемент на данный момент

if (right < n && arr[right] > arr[largest])

largest = right;

// Если наибольший элемент не является корнем

if (largest != i) {

swap(arr[i], arr[largest]);

// Рекурсивно вызываем heapify для поддерева, в котором был изменен корень

heapify(arr, n, largest);

}

}

// Функция сортировки кучей

void heapSort(vector<int>& arr) {

int n = arr.size();

// Построение кучи (перемещение наибольших элементов в конец массива)

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

heapify(arr, n, i);

// Один за другим извлекаем элементы из кучи

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

// Перемещаем текущий корень в конец

swap(arr[0], arr[i]);

// Вызываем heapify на уменьшенной куче

heapify(arr, i, 0);

}

}

// Функция для вывода массива

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

for (int num : arr) {

cout << num << " ";

}

cout << endl;

}

// Функция юнит-тестов

void testHeapSort() {

setlocale(LC\_ALL, "RU");

// Набор тестовых массивов

vector<pair<string, vector<int>>> testCases = {

{"Пустой массив", {}},

{"Один элемент", {42}},

{"Уже отсортированный массив", {1, 2, 3, 4, 5}},

{"Обратно отсортированный массив", {5, 4, 3, 2, 1}},

{"Случайный массив", {3, 1, 4, 1, 5, 9, 2, 6, 5}},

{"Массив с повторяющимися элементами", {10, 7, 7, 3, 2, 2, 5, 7}}

};

for (size\_t i = 0; i < testCases.size(); ++i) {

cout << "Тест " << i + 1 << ": " << testCases[i].first << endl;

vector<int> arr = testCases[i].second;

cout << "Исходный массив: ";

printArray(arr);

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

heapSort(arr);

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

auto duration = duration\_cast<microseconds>(end - start);

cout << "Отсортированный массив: ";

printArray(arr);

cout << "Время выполнения: " << duration.count() << " микросекунд" << endl;

// Проверка корректности сортировки

assert(is\_sorted(arr.begin(), arr.end()));

cout << "Тест успешно пройден!" << endl << endl;

}

}

int main() {

// Вызов юнит-тестов

testHeapSort();

return 0;

}

ПРИЛОЖЕНИЕ В

Листинг кода файла Bucket sort.cpp

#include <iostream>

#include <vector>

#include <algorithm>

#include <chrono>

#include <cmath>

#include <cassert>

#include <iomanip>

using namespace std;

using namespace std::chrono;

void bucketSort(vector<int>& arr, int k) {

int n = arr.size();

if (n == 0) return;

// Создаем k ведер

vector<vector<int>> buckets(k);

// Находим минимальный и максимальный элемент

int minValue = \*min\_element(arr.begin(), arr.end());

int maxValue = \*max\_element(arr.begin(), arr.end());

// Если все элементы массива одинаковы, массив уже отсортирован

if (minValue == maxValue) {

return;

}

// Распределяем элементы в ведра

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

int bucketIndex = ((arr[i] - minValue) \* (k - 1)) / (maxValue - minValue); // Индекс ведра

buckets[bucketIndex].push\_back(arr[i]);

}

// Сортируем элементы в каждом ведре

for (int i = 0; i < k; i++) {

sort(buckets[i].begin(), buckets[i].end());

}

// Объединяем отсортированные элементы из ведер

int index = 0;

for (int i = 0; i < k; i++) {

for (int num : buckets[i]) {

arr[index++] = num;

}

}

}

// Функция для вывода массива

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

for (int num : arr) {

cout << num << " ";

}

cout << endl;

}

// Юнит-тесты

void testBucketSort() {

const int defaultK = 5; // Значение по умолчанию для количества ведер

setlocale(LC\_ALL, "RU");

// Тестовые данные

vector<pair<string, vector<int>>> testCases = {

{"Пустой массив", {}},

{"Один элемент", {42}},

{"Уже отсортированный массив", {1, 2, 3, 4, 5}},

{"Обратно отсортированный массив", {5, 4, 3, 2, 1}},

{"Случайный массив", {3, 1, 4, 1, 5, 9, 2, 6, 5}},

{"Массив с повторяющимися элементами", {10, 7, 7, 3, 2, 2, 5, 7}}

};

for (size\_t i = 0; i < testCases.size(); ++i) {

cout << "Тест " << i + 1 << ": " << testCases[i].first << endl;

vector<int> arr = testCases[i].second;

cout << "Исходный массив: ";

printArray(arr);

// Определяем количество ведер

int k = defaultK;

if (!arr.empty()) {

k = max(1, min((int)sqrt(arr.size()), (int)arr.size()));

}

// Замер времени выполнения

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

bucketSort(arr, k);

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

auto duration = duration\_cast<microseconds>(end - start);

// Вывод результата

cout << "Отсортированный массив: ";

printArray(arr);

cout << "Количество ведер: " << k << endl;

cout << "Время выполнения: " << duration.count() << " микросекунд" << endl;

// Проверка корректности сортировки

assert(is\_sorted(arr.begin(), arr.end()));

cout << "Тест успешно пройден!" << endl << endl;

}

}

int main() {

// Запуск юнит-тестов

testBucketSort();

return 0;

}