МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ автономное ОБРАЗОВАТЕЛЬНОЕ

УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

МОСКОВСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

(ФГБОУ ВПО МПУ)



Кафедра СМАРТ-технологии

Лабораторная работа № 1

«Анализ сложности алгоритмов»

По дисциплине «Алгоритмы и структуры данных в робототехнике»

Группа \_\_\_\_241-324 \_\_\_\_\_\_

№ группы

Студент \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Сальников Л.В.

Подпись студента

Дата \_\_\_18.04.2025\_\_\_\_

Дата сдачи

Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Петрунина Е.В.

Подпись преподавателя

Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Клецкин А.А.

Подпись преподавателя

2025

Цель:

 Изучение и применение методов анализа сложности алгоритмов.

Выполнение задания:

Первым шагом было написать код для трех методов сортировки, а именно, сортировки пузырьком, сортировки вставками и сортировки слиянием.

Чтобы написать код сортировки необходимо понимать, как они работают в теории.

**Сортировка пузырьком**: алгоритм сортировки, который многократно проходит по списку, сравнивает соседние элементы и **меняет их местами**, если они стоят в **неправильном порядке** (например, если слева стоит большее число, чем справа).

Блок-схема работы алгоритма представлена на рисунке 1.

Теоретическая сложность: в лучшем случае O(n), если массив отсортирован. Худший и средний случай совпадают: O(n2).

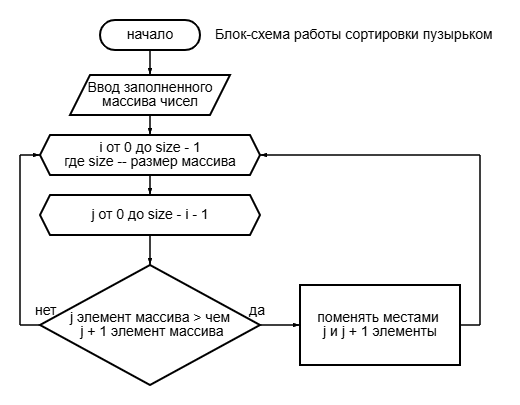


Рисунок 1 Сортировка пузырьком

**Сортировка вставками**: алгоритм, который строит отсортированный массив **поэлементно**: берёт по одному элементу из исходного массива и **вставляет его в нужное место** внутри уже отсортированной части массива.

Блок-схема представлена на рисунке 2.

Теоретическая сложность: в лучшем случае O(n), если массив уже отсортирован. Худший и средний случаи: O(n²)

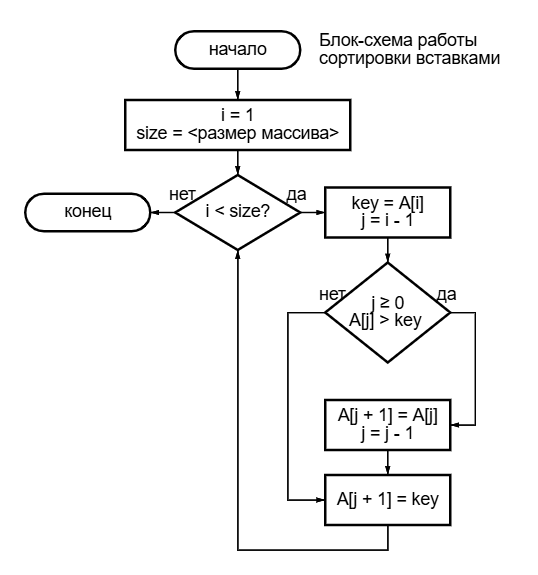


Рисунок 2 Сортировка вставками

**Сортировка слиянием:** рекурсивный алгоритм, который делит массив пополам, сортирует каждую половину и затем сливает их обратно в один отсортированный массив. Работает по принципу «разделяй и властвуй».

Блок-схема представлена на рисунке 3.

Теоретическая сложность O(n log n) в любом случае.

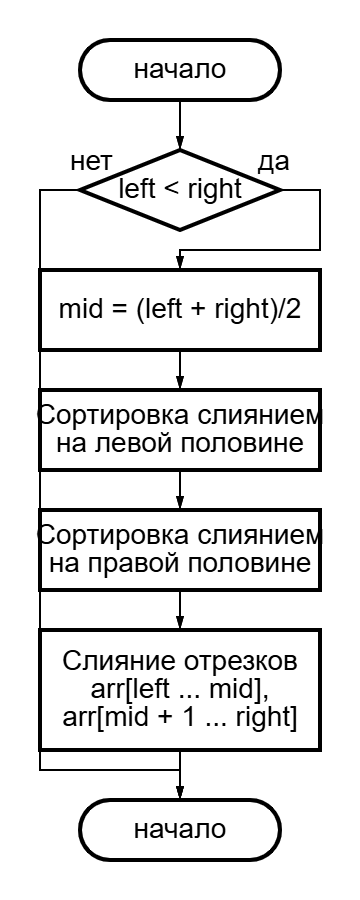


Рисунок 3 Сортировка слиянием

Следующим шагом написал код-шаблон для оценки времени выполнения сортировки каждым из методов. Код приведен ниже:

template<typename Func>

long long measureTime(Func sortFunction, vector<int> arr) {

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

sortFunction(arr);

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

auto duration = chrono::duration\_cast<chrono::milliseconds>(end - start);

return duration.count();

}

Данный шаблон находит время выполнения сортировки (берет начало и вычитает конец, тем самым находя время выполнения).

Далее, в главной функции создал три массива чисел размерами 100, 1000 и 10000 соответственно, заполняя их случайными числами от 0 до 10000 (для заполнения создал отдельную функцию, код которой привел ниже).

vector<int> generateRandomArray(int size)

{

vector<int> arr(size);

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

arr[i] = rand() % 10000;

return arr;

}

Далее в главной функции также прописал цикл, в котором выполняется поочередный проход всеми циклами всех методов сортировки, и вывод времени, затраченного на выполнение сортировки.

Полный код проекта привожу ниже:

#include <iostream>

#include <windows.h>

#include <vector>

#include <cstdlib> // для rand()

#include <ctime> // для time()

#include <chrono> // для измерения времени

using namespace std;

// Функция для генерации и заполнения массива необходимой длинны случайными числами

vector<int> generateRandomArray(int size)

{

vector<int> arr(size);

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

arr[i] = rand() % 10000;

return arr;

}

void bubbleSort(vector<int>& arr) // Сортировка пузырьком: проходит по массиву и меняет местами соседние элементы, если они идут в неправильном порядке. Повторяет это, пока массив не отсортирован.

{

int size = arr.size();

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

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

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

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

}

}

}

void insertionSort(vector<int>& arr) // Сортировка вставками: берет элементы один за другим и вставляет их в нужное место среди уже отсортированных элементов слева.

{

int size = arr.size();

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

int key = arr[i];

int j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = key;

}

}

void merge(vector<int>& arr, int left, int mid, int right) //Вспомогательная функция для сортировки слиянием

{

int i = left, j = mid + 1;

vector<int> temp;

while (i <= mid && j <= right) {

if (arr[i] < arr[j])

temp.push\_back(arr[i++]);

else

temp.push\_back(arr[j++]);

}

while (i <= mid) temp.push\_back(arr[i++]);

while (j <= right) temp.push\_back(arr[j++]);

for (int k = 0; k < temp.size(); ++k)

arr[left + k] = temp[k];

}

void mergeSort(vector<int>& arr, int left, int right) // Сортировка слиянием: делит массив на две части, сортирует каждую часть отдельно (рекурсивно), а потом сливает их обратно в один отсортированный массив.

{

if (left < right) {

int mid = (left + right) / 2;

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

}

template<typename Func>

long long measureTime(Func sortFunction, vector<int> arr) {

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

sortFunction(arr);

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

auto duration = chrono::duration\_cast<chrono::milliseconds>(end - start);

return duration.count();

}

int firstLR()

{

SetConsoleOutputCP(1251);

srand(time(nullptr)); // Инициализация генератора случайных чисел

vector<int> sizes = { 100, 1000, 10000 }; // Размеры массивов

for (int size : sizes) {

cout << "\nРазмер массива: " << size << "\n";

auto base = generateRandomArray(size);

// Bubble Sort

long long t1 = measureTime(bubbleSort, base);

cout << "Bubble Sort: " << t1 << " мс\n";

// Insertion Sort

long long t2 = measureTime(insertionSort, base);

cout << "Insertion Sort: " << t2 << " мс\n";

// Merge Sort (через лямбду)

auto mergeWrapper = [&](vector<int> a) {

mergeSort(a, 0, a.size() - 1);

};

long long t3 = measureTime(mergeWrapper, base);

cout << "Merge Sort: " << t3 << " мс\n" << endl;

}

return 0;

}

Эмпирический анализ

Следующим шагом провел трижды запуск функции, из чего имею:

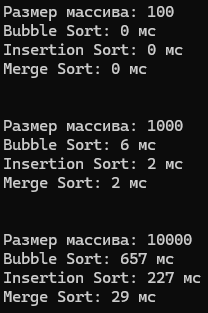
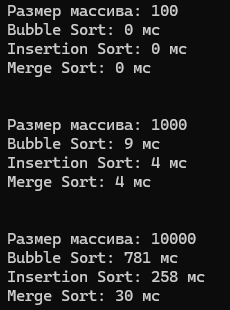
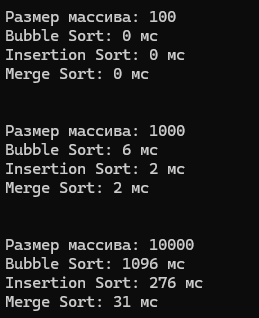


Рисунок 4 Рисунок 5 Рисунок 6

При нынешнем уровне вычислительных мощностей все три алгоритма справились с сортировкой ста элементов меньше чем за миллисекунду. С массивом в тысячу элементов сортировка пузырьком справляется хуже, сортировка вставками и слиянием отрабатывают за одно время. На массиве размером десять тысяч лучше всего видно разрыв в скорости выполнения: быстрее всех справляется сортировка слиянием, чуть медленнее отрабатывает сортировка вставками, медленнее всех работает сортировка пузырьком.

Теоретическая сложность

1. Сортировка пузырьком

|  |  |  |
| --- | --- | --- |
| **Сценарий** | **Кол-во операций** | **Оценка сложности** |
| Лучшая (уже отсортирован) | O(n) | ✅ |
| Средняя | O(n²) | ❌ |
| Худшая | O(n²) | ❌ |

1. Сортировка вставками

|  |  |  |
| --- | --- | --- |
| **Сценарий** | **Кол-во операций** | **Оценка сложности** |
| Лучшая (уже отсортирован) | O(n) | ✅ |
| Средняя | O(n²) | ❌ |
| Худшая | O(n²) | ❌ |

1. Сортировка слиянием

|  |  |  |
| --- | --- | --- |
| **Сценарий** | **Кол-во операций** | **Оценка сложности** |
| Лучшая | O(n log n) | ✅ |
| Средняя | O(n log n) | ✅ |
| Худшая | O(n log n) | ✅ |

И того имеем:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Алгоритм** | **Лучшая** | **Средняя** | **Худшая** | **Стабильность** | **Подходит для больших данных** |
| Пузырьком | O(n) | O(n²) | O(n²) | ✅ Да | ❌ Нет |
| Вставками | O(n) | O(n²) | O(n²) | ✅ Да | ❌ Нет |
| Слиянием | O(n log n) | O(n log n) | O(n log n) | ✅ Да | ✅ Да |

Сравнение теории и практики

Далее приведены два графика, показывающие зависимость времени от количества элементов, как получилось в действительности (средние значения) в миллисекундах и теоретическая сложность без единиц измерений.

Как можно видеть графики схожи, следовательно теория подтверждается на практике.

Вывод

В ходе лабораторной работы были реализованы и протестированы три алгоритма сортировки: пузырьком, вставками и слиянием. Эмпирические замеры времени показали, что сортировки пузырьком и вставками значительно уступают по производительности сортировке слиянием на больших массивах.

Теоретическая оценка сложности алгоритмов подтвердилась экспериментально:

* Пузырьком и вставками: O(n²) в среднем и худшем случаях,
* Слиянием: O(n log n) при любых входных данных.

Для работы с большими массивами следует использовать более эффективные алгоритмы, такие как сортировка слиянием.