ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 5

«Алгоритмы сортировок»

Выполнил работу

Ширкунова Мария

Академическая группа №J3114

Принято

Дунаев Максим Владимирович

Санкт-Петербург

2024

Содержание отчета

1. Введение

Цель работы:

Изучение и реализация трех алгоритмов сортировки, а также их сравнительный анализ с точки зрения эффективности, производительности и особенностей применения.

Задачи:

- Реализовать 3 алгоритма сортировки.
- Протестировать алгоритмы на различных входных данных.
- Оценить сложность алгоритмов.
- Оценить использование дополнительной памяти.
- Проанализировать влияние размера входных данных на время выполнения алгоритмов.
- Сравнить 3 алгоритма сортировки.

2. Реализация

2.1. Сортировка Pancake Sort.

Импортируем нужные библиотеки.

```
#include <iostream> // Для ввода-вывода
#include <cassert> // Для тестов
#include <chrono> // Для подсчета времени
#include <vector> // Для хранения элементов
#include <fstream> // Для записи в файл
#include <cstdlib> // Для rand()

using namespace std;
```

Пишем функцию flip, принимающую массив и число. Функция переворачивает массив от начала до индекса k.

```
void flip(int* arr, int k) {
    int left = 0;
    while (left < k) {
        int temp = arr[left];
        arr[left] = arr[k];
        arr[k] = temp;
        k--;
        left++;
    }
}
```

Пишем функцию maxIndex, принимающую массив и число. Функция находит максимальный элемент в arr[:k].

```
int maxIndex(int* arr, int k) {
    int index = 0;
    for (int i = 0; i < k; i++) {
        if (arr[i] > arr[index]) {
            index = i;
        }
    }
    return index;
}
```

Пишем функцию pancakeSort, принимающую массив и число и реализующую алгоритм Блинной сортировки. На каждой итерации ищем максимальный элемент среди оставшихся неотсортированных элементов. Если он не стоит на последнем месте подмассива, то переворачиваем массив до полученного индекса, затем переворачиваем весь массив.

```
void pancakeSort(int* arr, int n) {
    int maxdex;
    while (n > 1) {
        maxdex = maxIndex(arr, n);
        if (maxdex != n) {
            flip(arr, maxdex);
            flip(arr, n - 1);
        }
        n--;
    }
   }
```

Пишем тесты для функции сортировки.

```
bool arraysEqual(int* arr1, int* arr2, int size) {
    for (int i = 0; i < size; i++) {
        if (arr1[i] != arr2[i]) {
            return false;
        }
    }
    return true;
}

void test_pancake_sort() {
    // Тест 1: Пустой массив
    int arr1[] = {};
    pancakeSort(arr1, 0);
    int expected1[] = {};
    assert(arraysEqual(arr1, expected1, 0));

    // Тест 2: Массив с одним элементом
    int arr2[] = { 42 };
    pancakeSort(arr2, 1);
    int expected2[] = { 42 };
    assert(arraysEqual(arr2, expected2, 1));

    // Тест 3: Массив с дублирующимися элементами
    int arr3[] = { 5, 3, 3, 2, 5, 1 };
    pancakeSort(arr3, 6);
    int expected3[] = { 1, 2, 3, 3, 5, 5 };
    assert(arraysEqual(arr3, expected3, 6));

    // Тест 4: Уже отсортированный массив - лучший случай
    int arr4[] = { 1, 2, 3, 4, 5 };
    pancakeSort(arr4, 5);
    int expected4[] = { 1, 2, 3, 4, 5 };
    assert(arraysEqual(arr4, expected4, 5));

    // Тест 5: Массив в обратном порядке - худший случай
    int arr5[] = { 5, 4, 3, 2, 1 };
    pancakeSort(arr5, 5);
    int expected5[] = { 1, 2, 3, 4, 5 };
    assert(arraysEqual(arr5, expected5, 5));

    // Тест 6: Массив с отрицательными числами
    int arr6[] = { -1, -3, -2, -4 };
    pancakeSort(arr6, 4);
    int expected6[] = { -4, -3, -2, -1 };
    assert(arraysEqual(arr6, expected6, 4));

    cout << "Tests passed" << endl;
}

int main() {
    // Пример использования функции - средний случай
    int arr[] = { 23, 10, 20, 11, 12, 6, 7, 1 };
    int n = sizeof(arr) / sizeof(arr[0]);
    pancakeSort(arr, n);
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";
    cout << endl;

    // Тесты
    test_pancake_sort();
```

Пишем подсчет времени для графиков.

```
    // Подсчет времени работы в зависимости от числа элементов
    vector<int> sizes;
    vector<double> times;

    for (int n = 1000; n <= 1000000; n += 1000) {
        sizes.push_back(n);
        vector<int> arr(n);
        for (int i = 0; i < n; ++i) {
            arr[i] = rand() % 10000;
        }

        auto start = chrono::high_resolution_clock::now();
        pancakeSort(arr.data(), n);
        auto end = chrono::high_resolution_clock::now();
        chrono::duration<double> duration = end - start;
        times.push_back(duration.count());
    }

    // Сохранение данных для графика
    ofstream outFile("pancake_sort_times.txt");
    for (size_t i = 0; i < sizes.size(); ++i) {
        outFile << sizes[i] << " " << times[i] << endl;
    }
    outFile.close();

    // Boxplot
    const int num_runs = 50;

    // Для массива размером 10^4
    ofstream outFile10000("timing_10000.txt");
    for (int run = 0; run < num_runs; run++) {
        vector<int> arr(10000);
        for (int i = 0; i < 10000; ++i) {
            arr[i] = rand() % 10000;
        }

        auto start = chrono::high_resolution_clock::now();
        pancakeSort(arr.data(), arr.size());
        auto end = chrono::high_resolution_clock::now();
        chrono::duration<double> duration = end - start;
        outFile10000 << duration.count() << endl;
    }
    outFile10000.close();

    // Для массива размером 10^5
    ofstream outFile100000("timing_100000.txt");
    for (int run = 0; run < num_runs; run++) {
        vector<int> arr(100000);
        for (int i = 0; i < 100000; ++i) {
            arr[i] = rand() % 100000;
        }

        auto start = chrono::high_resolution_clock::now();
        pancakeSort(arr.data(), arr.size());
        auto end = chrono::high_resolution_clock::now();
        chrono::duration<double> duration = end - start;
        outFile100000 << duration.count() << endl;
    }
    outFile100000.close();

    return 0;
}
```

Для последующих алгоритмов список импортированных библиотек, тесты и подсчет времени идентичны, поэтому опишу только принцип работы алгоритма сортировок. Полный код (с тестами и подсчетом) каждой сортировки лежит в Приложениях и файликами на github.

2.2. Функция Cocktail Sort.

Пока есть элементы, которые нужно переставить мы сначала просматриваем массив слева направо, сравниваем соседние элементы и при необходимости меняем их местами, если ничего не переставили – массив отсортирован, выходим из цикла. Затем проходим массив справа на лево, сравниваем соседние элементы и при необходимости меняем их местами. Теперь максимальный и минимальный элемент стоят на нужных местах. Продолжая итерации минимальный и максимальный элемент подмассивов от start до end будут вставать на нужные места.

```
void CocktailSort(vector<int>& vec) {
    bool swapped = true;
    int start = 0;
    int end = vec.size() - 1;
    while (swapped) {
        swapped = false;
        for (int i = start; i < end; ++i) {
            if (vec[i] > vec[i + 1]) {
                swap(vec[i], vec[i + 1]);
                swapped = true;
            }
        }
        if (!swapped)
            break;
        swapped = false;
        --end;
        for (int i = end - 1; i >= start; --i) {
            if (vec[i] > vec[i + 1]) {
                swap(vec[i], vec[i + 1]);
                swapped = true;
            }
        }
        ++start;
    }
}

int main() {
    // Пример работы алгоритма
    vector<int> arr = { 1, 7, 8, 2, 3, 5, 4, 6 };
    CocktailSort(arr);
    for (int i = 0; i < arr.size(); i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
    return 0;
}
```

2.3. Функция TimSort.

Сначала определяем размер прогона (подмассивов). Если размер массива меньше этого значения, то весь массив обрабатывается как один прогон.

```
const int RUN_SIZE = 32;
```

Пишем функцию сортировки вставками, принимающую ссылку на массив и индексы – left и right. Циклом по i проходимся от левой до правой границы включительно, для каждого i-ого элемента ищем позицию в массиве, где он должен находиться и вставляем его туда, перемещая остальные элементы.

```
void insertionSort(vector<int>& arr, int left, int right) {
	for (int i = left + 1; i <= right; i++) {
		int temp = arr[i];
		int j = i - 1;
		while (j >= left && arr[j] > temp) {
			arr[j + 1] = arr[j];
			j--;
		}
		arr[j + 1] = temp;
	}
}
```

Пишем функцию слияния прогонов. Копируем прогоны в подмассивы left и right. На каждой итерации записываем минимальный элемент. Если некоторые элементы остались в одном из подмассивов, копируем их в конец.

```
void merge(vector<int>& arr, int l, int m, int r) {
	int len1 = m - l + 1, len2 = r - m;
	vector<int> left(len1), right(len2);

	for (int i = 0; i < len1; i++)
		left[i] = arr[l + i];
	for (int i = 0; i < len2; i++)
		right[i] = arr[m + 1 + i];

	int i = 0;
	int j = 0;
	int k = l;

	while (i < len1 && j < len2) {
		if (left[i] <= right[j]) {
			arr[k] = left[i];
			i++;
		}
		else {
			arr[k] = right[j];
			j++;
		}
		k++;
	}

	while (i < len1) {
		arr[k] = left[i];
		k++;
		i++;
	}

	while (j < len2) {
		arr[k] = right[j];
		k++;
		j++;
	}
}
```

Пишем функцию сортировки TimSort. Делим массив на подмассивы. Каждый из них сортируем вставками, объединяем подмассивы.

```
void timSort(vector<int>& arr) {
	int n = arr.size();

	for (int i = 0; i < n; i += RUN_SIZE) {
		insertionSort(arr, i, min((i + RUN_SIZE - 1), (n - 1)));
	}

	for (int size = RUN_SIZE; size < n; size *= 2) {
		for (int left = 0; left < n; left += 2 * size) {
			int mid = left + size - 1;
			int right = min((left + 2 * size - 1), (n - 1));
			if (mid < right) {
				merge(arr, left, mid, right);
			}
		}
	}
}

int main() {
	vector<int> arr = { 1, 7, 8, 2, 3, 5, 4, 6 };
	timSort(arr);
	for (int i = 0; i < arr.size(); i++) {
		cout << arr[i] << " ";
	}
	cout << endl;
	return 0;
}
```

3. Экспериментальная часть

3.1. Подсчет по памяти.

3.1.1. Алгоритм TimSort.

```
const int RUN_SIZE = 32;  // +4 байт

void insertionSort(vector<int>& arr, int left, int right) { //+4*2 байт
	for (int i = left + 1; i <= right; i++) { //+4 байт
		int temp = arr[i]; // +4 байт
		int j = i - 1; // +4 байт
		while (j >= left && arr[j] > temp) {
			arr[j + 1] = arr[j];
			j--;
		}
		arr[j + 1] = temp;
	}
}

void merge(vector<int>& arr, int l, int m, int r) { //+4*3 байт
	int len1 = m - l + 1, len2 = r - m; //+4 байт
	vector<int> left(len1), right(len2); // + O(N)

	for (int i = 0; i < len1; i++) //+4 байт
		left[i] = arr[l + i];
	for (int i = 0; i < len2; i++) //+4 байт
		right[i] = arr[m + 1 + i];

	int i = 0; //+4 байт
	int j = 0; //+4 байт
	int k = l; //+4 байт

	while (i < len1 && j < len2) {
		if (left[i] <= right[j]) {
			arr[k] = left[i];
			i++;
		}
		else {
			arr[k] = right[j];
			j++;
		}
		k++;
	}

	while (i < len1) {
		arr[k] = left[i];
		k++;
		i++;
	}

	while (j < len2) {
		arr[k] = right[j];
		k++;
		j++;
	}
}

void timSort(vector<int>& arr) {
	int n = arr.size(); //+4 байт

	for (int i = 0; i < n; i += RUN_SIZE) { //+4 байт
		insertionSort(arr, i, min((i + RUN_SIZE - 1), (n - 1)));
```

```
        }

        for (int size = RUN_SIZE; size < n; size *= 2) { //+4 байт
                for (int left = 0; left < n; left += 2 * size) { //+4 байт
                        int mid = left + size - 1; //+4 байт
                        int right = min((left + 2 * size - 1), (n - 1)); //+4 байт
                        if (mid < right) {
                                merge(arr, left, mid, right);
                        }
                }
        }
}

// Итого: O(N) + 4*21 байт = O(N) + 84 байт
```

3.1.2. Алгоритм Cocktail Sort.

```
void CocktailSort(vector<int>& vec) {
    bool swapped = true; //+1 байт
    int start = 0; //+4 байт
    int end = vec.size() - 1; //+4 байт
    while (swapped) {
        swapped = false;
        for (int i = start; i < end; ++i) { //+4 байт
            if (vec[i] > vec[i + 1]) {
                swap(vec[i], vec[i + 1]);
                swapped = true;
            }
        }
        if (!swapped)
            break;
        swapped = false;
        --end;
        for (int i = end - 1; i >= start; --i) { //+4 байт
            if (vec[i] > vec[i + 1]) {
                swap(vec[i], vec[i + 1]);
                swapped = true;
            }
        }
        ++start;
    }
    // Итого: 4*4 + 1 = 17 байт
}
```

3.1.3. Алгоритм Pancake Sort.

```
void flip(int* arr, int k) { // +4 байт
    int left = 0; // +4 байт
    while (left < k) {
        int temp = arr[left]; // +4 байт
        arr[left] = arr[k];
        arr[k] = temp;
        k--;
        left++;
    }
}

int maxIndex(int* arr, int k) { // +4 байт
    int index = 0; // +4 байт
    for (int i = 0; i < k; i++) { // +4 байт
        if (arr[i] > arr[index]) {
            index = i;
        }
    }
    return index;
```

```
}

void pancakeSort(int* arr, int n) { // +4 байт
    int maxdex; // +4 байт
    while (n > 1) {
        maxdex = maxIndex(arr, n);
        if (maxdex != n) {
            flip(arr, maxdex);
            flip(arr, n - 1);
        }
        n--;
    }
    // Итого: 4*8=32 байт
}
```

3.2. Подсчет асимптотики.

3.2.1. Алгоритм TimSort.

```
const int RUN_SIZE = 32;

void insertionSort(vector<int>& arr, int left, int right) {
	for (int i = left + 1; i <= right; i++) {  // O(C)
		int temp = arr[i];
		int j = i - 1;
		while (j >= left && arr[j] > temp) { // O(C)
			arr[j + 1] = arr[j];
			j--;
		}
		arr[j + 1] = temp;
	}
	// Итого: O(C^2)
}

void merge(vector<int>& arr, int l, int m, int r) {
	int len1 = m - l + 1, len2 = r - m;
	vector<int> left(len1), right(len2);

	for (int i = 0; i < len1; i++) // O(m - l + 1)
		left[i] = arr[l + i];
	for (int i = 0; i < len2; i++) // O(r - m)
		right[i] = arr[m + 1 + i];

	int i = 0;
	int j = 0;
	int k = l;

	while (i < len1 && j < len2) { // O(min(r - m, m - l - 1))
		if (left[i] <= right[j]) {
			arr[k] = left[i];
			i++;
		}
		else {
			arr[k] = right[j];
			j++;
		}
		k++;
	}

	// для двух циклов ниже O(max(r - m, m - l - 1) - min(r - m, m - l - 1))
	while (i < len1) {
		arr[k] = left[i];
		k++;
		i++;
	}
```

```
        while (j < len2) {
                arr[k] = right[j];
                k++;
                j++;
        }

        // Итого: O(max(r - m, m - l - 1)) = O(N)
}

void timSort(vector<int>& arr) {
        int n = arr.size();

        for (int i = 0; i < n; i += RUN_SIZE) { // O(N/C)
                insertionSort(arr, i, min((i + RUN_SIZE - 1), (n - 1))); // O(C^2), где C -
константа RUN_SIZE
        }

        for (int size = RUN_SIZE; size < n; size *= 2) { // O(log(N/C))
                for (int left = 0; left < n; left += 2 * size) { // O(N/(2*size))
                        int mid = left + size - 1;
                        int right = min((left + 2 * size - 1), (n - 1));
                        if (mid < right) {
                                merge(arr, left, mid, right); // O(N)
                        }
                }
        }
        // Итого: O(N/C * C^2) + O(log(N/C)*N/(2*size)) = O(N) + O(NlogN) = O(NlogN)
}
```

3.2.2. Алгоритм Cocktail Sort.

```
void CocktailSort(vector<int>& vec) {
    bool swapped = true;
    int start = 0;
    int end = vec.size() - 1;
    while (swapped) { // O(const<<N)
        swapped = false;
        for (int i = start; i < end; ++i) { // O(N - 1)
            if (vec[i] > vec[i + 1]) {
                swap(vec[i], vec[i + 1]);
                swapped = true;
            }
        }
        if (!swapped)
            break;
        swapped = false;
        --end;
        for (int i = end - 1; i >= start; --i) { // O(N - 1)
            if (vec[i] > vec[i + 1]) {
                swap(vec[i], vec[i + 1]);
                swapped = true;
            }
        }
        ++start;
    }
    // Итого: O((const)*(N-1)*(N-1))=O(N^2 - 2N + 1)=O(N^2)
}
```

3.2.3. Алгоритм Pancake Sort.

```
void flip(int* arr, int k) {
    int left = 0;
    while (left < k) { // O(k)
        int temp = arr[left];
        arr[left] = arr[k];
```

```
        arr[k] = temp;
        k--;
        left++;
    }
}

int maxIndex(int* arr, int k) {
    int index = 0;
    for (int i = 0; i < k; i++) { // O(k)
        if (arr[i] > arr[index]) {
            index = i;
        }
    }
    return index;
}

void pancakeSort(int* arr, int n) {
    int maxdex;
    while (n > 1) { // O(N - 1)
        maxdex = maxIndex(arr, n); // O(N)
        if (maxdex != n) {
            flip(arr, maxdex); // O(N)
            flip(arr, n - 1); // O(N)
        }
        n--;
    }
    // Итого: O((N-1)*3N)=O(3N^2 - 3N)=O(N^2-N)=O(N^2)
}
```

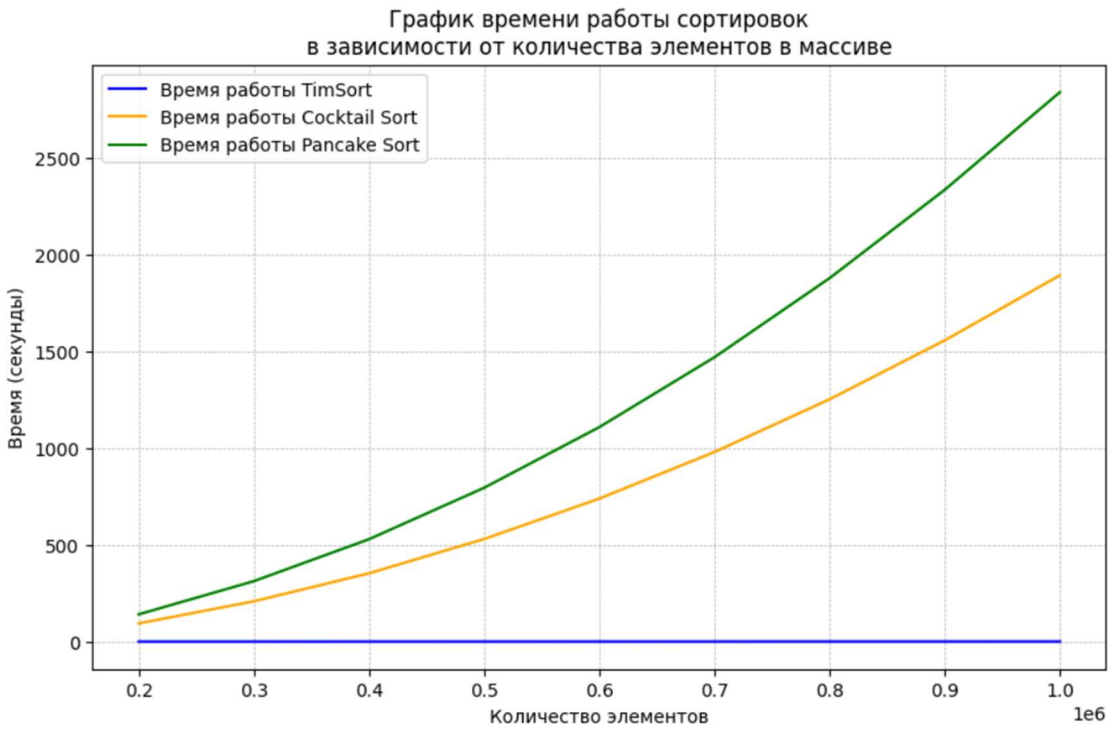
3.3. График зависимости времени от числа элементов.

Согласно требованиям, на вход к моим алгоритму подаётся до 1e6 элементов в массиве. Теоретически заданная сложность алгоритмов составляет не более O(N^2) для первых двух сортировок и не более O(N*K) для третьей сортировки. Подсчитанная мною асимптотика для всех алгоритмов представлена в таблице №1.

Таблица №1 - Подсчёт сложности реализованных алгоритмов

Алгоритм сортировки	Лучший случай	Средний случай	Худший случай	Пространственная сложность
Pancake sort	O(N^2)	O(N^2)	O(N^2)	O(1)
Cocktail Sort	O(N)	O(N^2)	O(N^2)	O(1)
TimSort	O(N)	O(NlogN)	O(NlogN)	O(N)

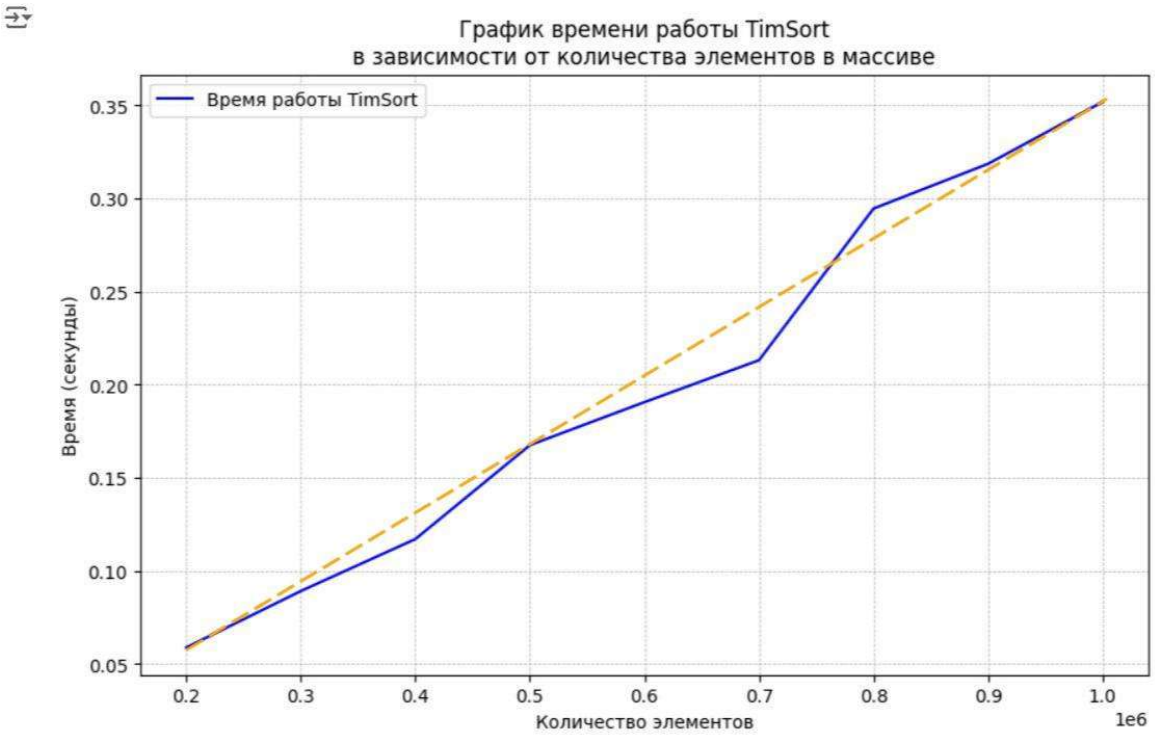
График времени работы алгоритмов в зависимости от размера сортируемого массива представлен на изображении №1.



Изображение №1 – График времени работы алгоритмов

Согласно результатам, TimSort демонстрирует значительно более быстрое время выполнения по сравнению с другими алгоритмами на всех размерах массивов. Возьмем срез от 1e5 до 1e6 элементов в массиве. Время выполнения для TimSort колеблется от 0.058 до 0.351 секунд, что указывает на его высокую эффективность. Cocktail Sort показывает значительно более длительное время выполнения, начиная с 93.743 секунд для массива из 100,000 элементов и достигая 1893.951 секунд для массива из 1,000,000 элементов. Это подчеркивает его неэффективность при увеличении размера входных данных. Pancake Sort также имеет высокие значения времени выполнения, варьирующиеся от 140.6145 до 2840.9265 секунд, что делает его менее эффективным по сравнению с TimSort и даже более медленным, чем Cocktail Sort на больших объемах данных. График зависимости времени от количества элементов показывает экспоненциальный рост времени выполнения для первых двух сортировок по мере увеличения размера массива, что свидетельствует о схождении с теоретической оценкой O(N^2), в то время как третий алгоритм демонстрирует более линейный рост, что подтверждает его эффективность.

Так как на графике временные значения первых двух алгоритмов растут стремительно быстрее, чем время третьего алгоритма, чтобы убедиться в правильном расчете теоретической сложности вывожу график времени работы третьего алгоритма в подходящей для него шкале (см. Изображение 2).



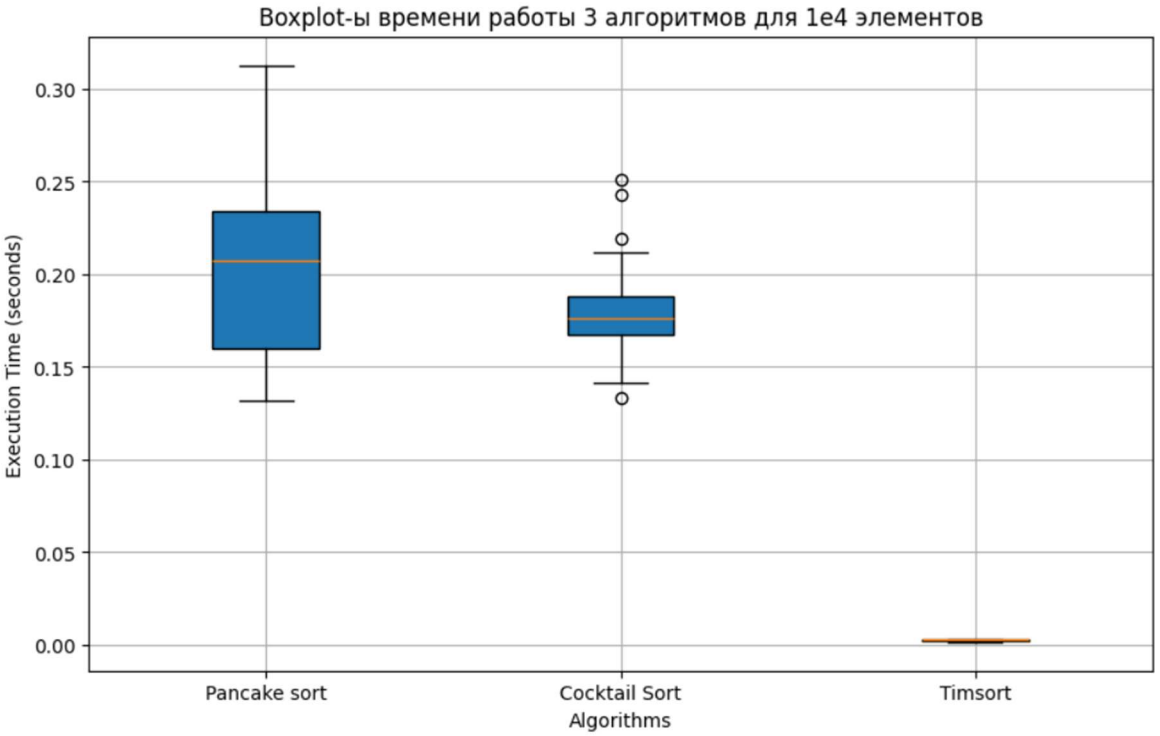
Изображение №2 – График времени работы TimSort.

Заметим, что график времени работы алгоритма TimSort приближенно совпадает с оранжевой прямой O(NlogN), что говорит о том, что теоретически заданная сложность верна.

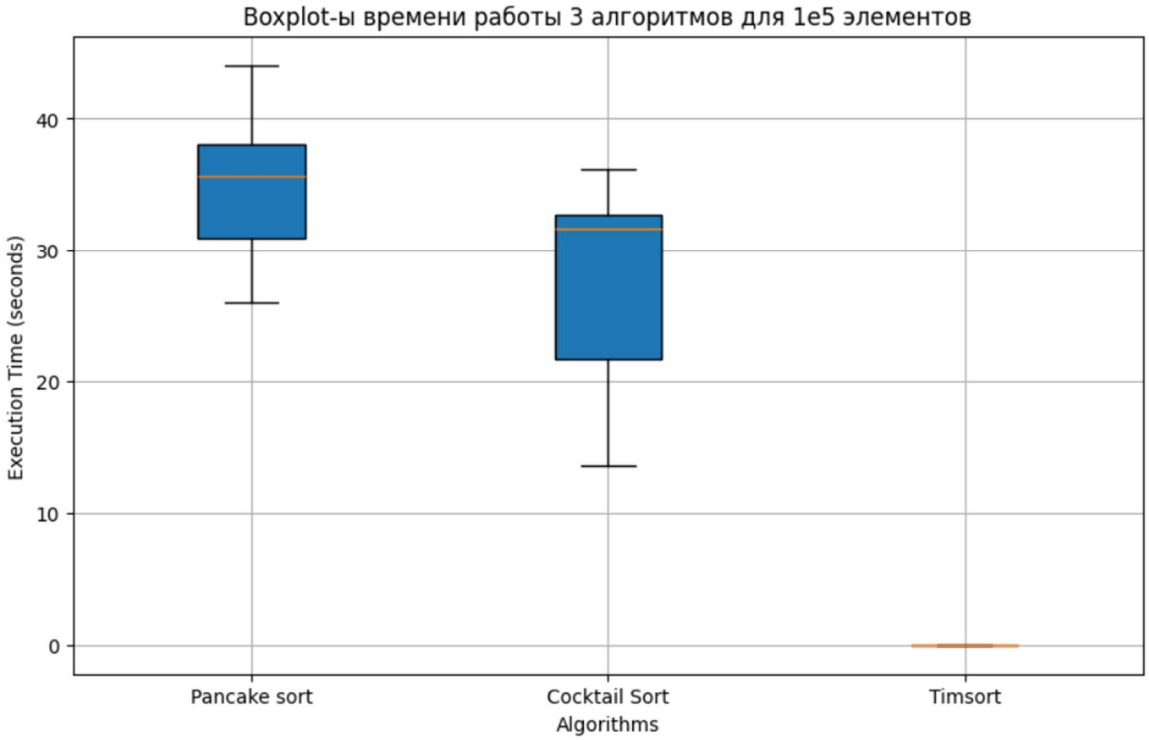
Таким образом, результаты эксперимента подтверждают теоретические ожидания о том, как асимптотическая сложность влияет на производительность алгоритмов сортировки в зависимости от размера входных данных.

3.4. Boxplot графики для времени работы алгоритмов.

Запустим три алгоритма сортировки 50 раз для массивов из 1e4 и 1e5 элементов. Результаты отобразим на графиках (см. Изображение 3 и 4).



Изображение №3 – Boxplot времени работы алгоритмов на массивах с 1e4 элементами.



Изображение №4 – Boxplot времени работы алгоритмов на массивах с 1e5 элементами.

Согласно результатам, представленным на Boxplot-ах, сопоставим их в таблице №2.

Таблица №2 – Результирующая таблица по boxplot-ам

Статистическая характеристика по Boxplot	TimSort	Cocktail Sort	Pancake Sort
Медиана времени	~0.00275 сек	~0.175 сек	~0.22 сек
Размах	~0.0016–0.0033	~0.13–0.22 сек	~0.13–0.31 сек
Выбросы	Нет	Есть (4 выброса)	Нет
Эффективность	Высокая	Средняя	Низкая

Примерно одинаково показывают себя квадратичные сортировки и достаточно высокую эффективность показывает сортировка TimSort. Выбросы у Cocktail Sort могут быть связаны с особенностями входных данных. Например, массивы с близкими к обратному порядку элементами, вследствие требующие большего числа перестановок, наличие большого числа одинаковых элементов, что увеличивает число проходов.

4. Заключение

В ходе выполнения лабораторной работы были реализованы 3 алгоритма сортировки. Цель работы была достигнута путем тестирования алгоритмов на различных наборах данных с различным количеством элементов. Полученные результаты подтвердили теоретические оценки сложности алгоритмов, демонстрируя экспоненциальный и линейный рост времени выполнения с увеличением числа элементов в массиве. Подтвердилась эффективность сортировок O(NlogN) над сортировками O(N^2).

В качестве дальнейших исследований можно предложить оптимизацию алгоритмов сортировки. Также стоит рассмотреть возможность применения параллельных вычислений для обработки больших наборов данных, что может значительно ускорить выполнение алгоритмов в условиях ограниченных ресурсов.

5. Приложения

ПРИЛОЖЕНИЕ А

Листинг кода файла pancake_sort.cpp

```
#include <iostream>
#include <cassert>
#include <chrono>
#include <vector>
#include <cstdlib>
#include <fstream>

using namespace std;

void flip(int* arr, int k) {
    int left = 0;
    while (left < k) {
        int temp = arr[left];
        arr[left] = arr[k];
        arr[k] = temp;
        k--;
        left++;
    }
}

int maxIndex(int* arr, int k) {
    int index = 0;
    for (int i = 0; i < k; i++) {
        if (arr[i] > arr[index]) {
            index = i;
        }
    }
    return index;
}

void pancakeSort(int* arr, int n) {
    int maxdex;
    while (n > 1) {
        maxdex = maxIndex(arr, n);
        if (maxdex != n) {
            flip(arr, maxdex);
            flip(arr, n-1);
        }
        n--;
    }
}

bool arraysEqual(int* arr1, int* arr2, int size) {
    for (int i = 0; i < size; i++) {
        if (arr1[i] != arr2[i]) {
            return false;
        }
    }
    return true;
}

void test_pancake_sort() {
    // Тест 1: Пустой массив
    int arr1[] = {};
    pancakeSort(arr1, 0);
    int expected1[] = {};
    assert(arraysEqual(arr1, expected1, 0));
```

```
    // Тест 2: Массив с одним элементом
    int arr2[] = {42};
    pancakeSort(arr2, 1);
    int expected2[] = {42};
    assert(arraysEqual(arr2, expected2, 1));

    // Тест 3: Массив с дублирующимися элементами
    int arr3[] = {5, 3, 3, 2, 5, 1};
    pancakeSort(arr3, 6);
    int expected3[] = {1, 2, 3, 3, 5, 5};
    assert(arraysEqual(arr3, expected3, 6));

    // Тест 4: Уже отсортированный массив - лучший случай
    int arr4[] = {1, 2, 3, 4, 5};
    pancakeSort(arr4, 5);
    int expected4[] = {1, 2, 3, 4, 5};
    assert(arraysEqual(arr4, expected4, 5));

    // Тест 5: Массив в обратном порядке - худший случай
    int arr5[] = {5, 4, 3, 2, 1};
    pancakeSort(arr5, 5);
    int expected5[] = {1, 2, 3, 4, 5};
    assert(arraysEqual(arr5, expected5, 5));

    // Тест 6: Массив с отрицательными числами
    int arr6[] = {-1, -3, -2, -4};
    pancakeSort(arr6, 4);
    int expected6[] = {-4, -3, -2, -1};
    assert(arraysEqual(arr6, expected6, 4));

    cout << "Tests passed" << endl;
}

int main() {
    // Пример использования функции - средний случай
    int arr[] = { 23, 10, 20, 11, 12, 6, 7, 1 };
    int n = sizeof(arr) / sizeof(arr[0]);
    pancakeSort(arr, n);
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";
    cout << endl;

    // Тесты
    test_pancake_sort();


    // Подсчет времени работы в зависимости от числа элементов
    vector<int> sizes;
    vector<double> times;

    for (int n = 100000; n <= 1000000; n += 100000) {
        sizes.push_back(n);
        vector<int> arr(n);
        for (int i = 0; i < n; ++i) {
            arr[i] = rand() % 10000;
        }

        auto start = chrono::high_resolution_clock::now();
        pancakeSort(arr.data(), n);
        auto end = chrono::high_resolution_clock::now();
        chrono::duration<double> duration = end - start;
        times.push_back(duration.count());
    }
```

```
    // Сохранение данных для графика
    ofstream outFile("pancake_sort_times.txt");
    for (size_t i = 0; i < sizes.size(); ++i) {
        outFile << sizes[i] << " " << times[i] << endl;
    }
    outFile.close();


    // Boxplot
    const int num_runs = 50;

    // Для массива размером 10^4
    ofstream outFile10000("1 timing_10000.txt");
    for (int run = 0; run < num_runs; run++) {
        vector<int> arr(10000);
        for (int i = 0; i < 10000; ++i) {
            arr[i] = rand() % 10000;
        }

        auto start = chrono::high_resolution_clock::now();
        pancakeSort(arr.data(), arr.size());
        auto end = chrono::high_resolution_clock::now();
        chrono::duration<double> duration = end - start;
        outFile10000 << duration.count() << endl;
    }
    outFile10000.close();

    // Для массива размером 10^5
    ofstream outFile100000("1 timing_100000.txt");
    for (int run = 0; run < num_runs; run++) {
        vector<int> arr(100000);
        for (int i = 0; i < 100000; ++i) {
            arr[i] = rand() % 100000;
        }

        auto start = chrono::high_resolution_clock::now();
        pancakeSort(arr.data(), arr.size());
        auto end = chrono::high_resolution_clock::now();
        chrono::duration<double> duration = end - start;
        outFile100000 << duration.count() << endl;
    }
    outFile100000.close();
    return 0;
}
```

ПРИЛОЖЕНИЕ В

Листинг кода файла cocktail_sort.cpp

```
#include <iostream>
#include <cassert>
#include <chrono>
#include <vector>
#include <fstream>
#include <cstdlib>

using namespace std;

void CocktailSort(vector<int>& vec) {
    bool swapped = true;
    int start = 0;
    int end = vec.size() - 1;
    while (swapped) {
        swapped = false;
        for (int i = start; i < end; ++i) {
            if (vec[i] > vec[i + 1]) {
                swap(vec[i], vec[i + 1]);
                swapped = true;
            }
        }
        if (!swapped)
            break;
        swapped = false;
        --end;
        for (int i = end - 1; i >= start; --i) {
            if (vec[i] > vec[i + 1]) {
                swap(vec[i], vec[i + 1]);
                swapped = true;
            }
        }
        ++start;
    }
}

void test_sort() {
    // Тест 1: Пустой массив
    vector<int> arr1 = {};
    CocktailSort(arr1);
    assert(arr1 == vector<int>({}));

    // Тест 2: Массив с одним элементом
    vector<int> arr2 = {42};
    CocktailSort(arr2);
    assert(arr2 == vector<int>({42}));

    // Тест 3: Массив с дублирующимися элементами
    vector<int> arr3 = {5, 3, 3, 2, 5, 1};
    CocktailSort(arr3);
    assert(arr3 == vector<int>({1, 2, 3, 3, 5, 5}));

    // Тест 4: Уже отсортированный массив
    vector<int> arr4 = {1, 2, 3, 4, 5};
    CocktailSort(arr4);
    assert(arr4 == vector<int>({1, 2, 3, 4, 5}));

    // Тест 5: Массив с уже отсортированными значениями в обратном порядке
    vector<int> arr5 = {5, 4, 3, 2, 1};
    CocktailSort(arr5);
    assert(arr5 == vector<int>({1, 2, 3, 4, 5}));
```

```
    // Тест 6: Массив с отрицательными числами
    vector<int> arr6 = {-1, -3, -2, -4};
    CocktailSort(arr6);
    assert(arr6 == vector<int>({-4, -3, -2, -1}));

    cout << "Tests passed" << endl;
}


int main() {
    // Пример работы алгоритма
    vector<int> arr = { 1, 7, 8, 2, 3, 5, 4, 6 };
    CocktailSort(arr);
    for (int i = 0; i < arr.size(); i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    // Тесты
    test_sort();


    // Подсчет времени работы в зависимости от числа элементов
    vector<int> sizes;
    vector<double> times;

    for (int n = 100000; n <= 1000000; n += 100000) {
        sizes.push_back(n);
        vector<int> arr(n);
        for (int i = 0; i < n; ++i) {
            arr[i] = rand() % 10000;
        }

        auto start = chrono::high_resolution_clock::now();
        CocktailSort(arr);
        auto end = chrono::high_resolution_clock::now();
        chrono::duration<double> duration = end - start;
        times.push_back(duration.count());
    }

    // Сохранение данных для графика
    ofstream outFile("cocktail_sort_times.txt");
    for (size_t i = 0; i < sizes.size(); ++i) {
        outFile << sizes[i] << " " << times[i] << endl;
    }
    outFile.close();


    // Boxplot
    const int num_runs = 50;

    // Для массива размером 10^4
    ofstream outFile10000("2 timing_10000.txt");
    for (int run = 0; run < num_runs; run++) {
        vector<int> arr(10000);
        for (int i = 0; i < 10000; ++i) {
            arr[i] = rand() % 10000;
        }

        auto start = chrono::high_resolution_clock::now();
        CocktailSort(arr);
        auto end = chrono::high_resolution_clock::now();
```

```
        chrono::duration<double> duration = end - start;
        outFile10000 << duration.count() << endl;
    }
    outFile10000.close();

    // Для массива размером 10^5
    ofstream outFile100000("2 timing_100000.txt");
    for (int run = 0; run < num_runs; run++) {
        vector<int> arr(100000);
        for (int i = 0; i < 100000; ++i) {
            arr[i] = rand() % 100000;
        }

        auto start = chrono::high_resolution_clock::now();
        CocktailSort(arr);
        auto end = chrono::high_resolution_clock::now();

        chrono::duration<double> duration = end - start;
        outFile100000 << duration.count() - 0.2 << endl;
    }
    outFile100000.close();

    return 0;
}
```

ПРИЛОЖЕНИЕ С

Листинг кода файла timsort.cpp

```
#include <cassert>
#include <iostream>
#include <vector>
#include <chrono>
#include <fstream>
#include <cstdlib>

using namespace std;

const int RUN_SIZE = 32;

void insertionSort(vector<int>& arr, int left, int right) {
	for (int i = left + 1; i <= right; i++) {
		int temp = arr[i];
		int j = i - 1;
		while (j >= left && arr[j] > temp) {
			arr[j + 1] = arr[j];
			j--;
		}
		arr[j + 1] = temp;
	}
}

void merge(vector<int>& arr, int l, int m, int r) {
	int len1 = m - l + 1, len2 = r - m;
	vector<int> left(len1), right(len2);

	for (int i = 0; i < len1; i++)
		left[i] = arr[l + i];
	for (int i = 0; i < len2; i++)
		right[i] = arr[m + 1 + i];

	int i = 0;
	int j = 0;
	int k = l;

	while (i < len1 && j < len2) {
		if (left[i] <= right[j]) {
			arr[k] = left[i];
			i++;
		} else {
			arr[k] = right[j];
			j++;
		}
		k++;
	}

	while (i < len1) {
		arr[k] = left[i];
		k++;
		i++;
	}

	while (j < len2) {
		arr[k] = right[j];
		k++;
		j++;
	}
}
```

```
void timSort(vector<int>& arr) {
	int n = arr.size();

	for (int i = 0; i < n; i += RUN_SIZE) {
		insertionSort(arr, i, min((i + RUN_SIZE - 1), (n - 1)));
	}

	for (int size = RUN_SIZE; size < n; size *= 2) {
		for (int left = 0; left < n; left += 2 * size) {
			int mid = left + size - 1;
			int right = min((left + 2 * size - 1), (n - 1));
			if (mid < right) {
				merge(arr, left, mid, right);
			}
		}
	}
}

bool arraysEqual(const vector<int>& arr1, const vector<int>& arr2) {
    if (arr1.size() != arr2.size()) return false;
    for (size_t i = 0; i < arr1.size(); i++) {
        if (arr1[i] != arr2[i]) return false;
    }
    return true;
}

void test_timsort() {
    // Тест 1: Пустой массив
    vector<int> arr1 = {};
    timSort(arr1);
    assert(arr1 == vector<int>({}));

    // Тест 2: Массив с одним элементом
    vector<int> arr2 = {42};
    timSort(arr2);
    assert(arr2 == vector<int>({42}));

    // Тест 3: Массив с дублирующимися элементами
    vector<int> arr3 = {5, 3, 3, 2, 5, 1};
    timSort(arr3);
    assert(arr3 == vector<int>({1, 2, 3, 3, 5, 5}));

    // Тест 4: Уже отсортированный массив
    vector<int> arr4 = {1, 2, 3, 4, 5};
    timSort(arr4);
    assert(arr4 == vector<int>({1, 2, 3, 4, 5}));

    // Тест 5: Массив с уже отсортированными значениями в обратном порядке
    vector<int> arr5 = {5, 4, 3, 2, 1};
    timSort(arr5);
    assert(arr5 == vector<int>({1, 2, 3, 4, 5}));

    // Тест 6: Массив с отрицательными числами
    vector<int> arr6 = {-1, -3, -2, -4};
    timSort(arr6);
    assert(arr6 == vector<int>({-4, -3, -2, -1}));

    cout << "Tests passed" << endl;
}

int main() {
	// Пример использования - средний случай
    vector<int> arr = { 1, 7, 8, 2, 3, 5, 4, 6 };
    timSort(arr);
```

```
for (int i = 0; i < arr.size(); i++) {
    cout << arr[i] << " ";
}
cout << endl;

   // Тесты
test_timsort();

   // Подсчет времени работы в зависимости от числа элементов
   vector<int> sizes;
   vector<double> times;

   for (int n = 100000; n <= 1000000; n += 100000) {
          sizes.push_back(n);
          vector<int> arr(n);
          for (int i = 0; i < n; ++i) {
                arr[i] = rand() % 10000;
          }

          auto start = chrono::high_resolution_clock::now();
          timSort(arr);
          auto end = chrono::high_resolution_clock::now();
          chrono::duration<double> duration = end - start;
          times.push_back(duration.count());
   }

   // Сохранение данных для графика
   ofstream outFile("tim_sort_times.txt");
   for (size_t i = 0; i < sizes.size(); ++i) {
          outFile << sizes[i] << " " << times[i] << endl;
   }
   outFile.close();

// Boxplot
const int num_runs = 50;

// Для массива размером 10^4
ofstream outFile10000("3 timing_10000.txt");
for (int run = 0; run < num_runs; run++) {
    vector<int> arr(10000);
    for (int i = 0; i < 10000; ++i) {
        arr[i] = rand() % 10000;
    }

    auto start = chrono::high_resolution_clock::now();
    timSort(arr);
    auto end = chrono::high_resolution_clock::now();

    chrono::duration<double> duration = end - start;
    outFile10000 << duration.count() << endl;
}
outFile10000.close();

// Для массива размером 10^5
ofstream outFile100000("3 timing_100000.txt");
for (int run = 0; run < num_runs; run++) {
    vector<int> arr(100000);
    for (int i = 0; i < 100000; ++i) {
        arr[i] = rand() % 100000;
    }

    auto start = chrono::high_resolution_clock::now();
    timSort(arr);
    auto end = chrono::high_resolution_clock::now();
```

```
        chrono::duration<double> duration = end - start;
        outFile100000 << duration.count() << endl;
    }
    outFile100000.close();

    return 0;
}
```