ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 5

«Алгоритмы сортировок»

Выполнил работу

Бровкин Аким

Академическая группа J3110

Принято

Ментор, Вершинин Владислав

Санкт-Петербург

2024

1. Введение

Цель данной работы заключается в реализации и сравнении производительности трех алгоритмов сортировки: Heap Sort, Bucket Sort и Tim Sort.  
Задачи, которые необходимо решить:

Реализовать указанные алгоритмы на языке C++.

Провести тестирование работы алгоритмов на различных наборах данных.

Оценить временную и пространственную сложность реализованных алгоритмов.

Построить график зависимости времени выполнения от размера входного массива.

Сравнить результаты с теоретическими оценками.

1. Теоретическая подготовка

**Heap Sort (Пирамидальная сортировка)**

**Тип данных**: массив.

**Алгоритм**: используется структура данных бинарная куча (max-heap).

**Сложность**:

**Временная**: O(n log n) на всех этапах сортировки.

**Пространственная**: O(1), так как сортировка выполняется на месте.

**Bucket Sort (Блочная сортировка)**

**Тип данных**: массив чисел с равномерным распределением.

**Алгоритм**: входной массив делится на несколько блоков, каждый блок сортируется (в нашем случае методом **Bubble Sort**).

**Сложность**:

**Средняя временная**: O(n + k), где k — количество элементов в блоках.

**Худшая временная**: O(n^2), если элементы неравномерно распределены.

**Пространственная**: O(n), дополнительная память для блоков.

**Tim Sort**

**Тип данных**: массив.

**Алгоритм**: гибридный метод, объединяющий **Insertion Sort** и **Merge Sort**.

**Сложность**:

**Временная**: O(n log n) в худшем случае.

**Пространственная**: O(n).

Реализация

В данной работе были реализованы три алгоритма сортировки: Heap Sort, Bucket Sort и Tim Sort. Рассмотрим процесс реализации каждого из них:

Heap\_Sort  
Алгоритм пирамидальной сортировки (Heap Sort) использует бинарную кучу для упорядочивания элементов. Сначала строится максимальная куча из исходного массива, затем элементы последовательно извлекаются из кучи и размещаются в отсортированной части массива. Основные этапы включают:

Формирование структуры кучи с помощью функции adjustHeap, которая рекурсивно корректирует поддерево.

Перемещение корневого элемента в конец массива и корректировка оставшейся части кучи.

Временная сложность алгоритма составляет O(nlog n), а пространственная сложность — O(1)

Bucket\_Sort  
Блочная сортировка (Bucket Sort) работает путём распределения элементов в несколько блоков (бакетов), сортировки каждого блока и объединения всех элементов обратно в массив. Реализация включает:

Создание массива блоков b[] на основе размера входных данных.

Распределение элементов по блокам с учётом их значений.

Сортировка элементов внутри каждого блока методом пузырьковой сортировки.

Объединение отсортированных блоков в исходный массив.

В худшем случае временная сложность составляет O(n^2), а пространственная сложность — O(n)

Tim\_Sort  
Tim Sort — это гибридный алгоритм, объединяющий преимущества сортировки вставками и сортировки слиянием. Алгоритм разбивает массив на небольшие подмассивы (раны), сортирует их методом вставок и объединяет с использованием сортировки слиянием. Реализация алгоритма состоит из следующих этапов:

Разделение массива на раны: Размер ранов выбирается на основе длины массива (обычно используется минимальный размер 32 или 64).

Сортировка ран методом вставок: Маленькие подмассивы эффективно сортируются с помощью сортировки вставками, поскольку этот метод показывает высокую производительность на небольших наборах данных.

Слияние ранов: Отсортированные раны объединяются попарно с помощью процедуры слияния, оптимизируя работу на основе их размеров.

Временная сложность алгоритма TimSort составляет O(nlog n) в худшем случае и O(n) при частично отсортированных данных. Пространственная сложность — O(n), так как при слиянии требуется дополнительная память для хранения временных массивов.

Реализация TimSort учитывает особенности данных и адаптируется к их структуре, что делает его эффективным в реальных задачах.

4. Экспериментальная часть

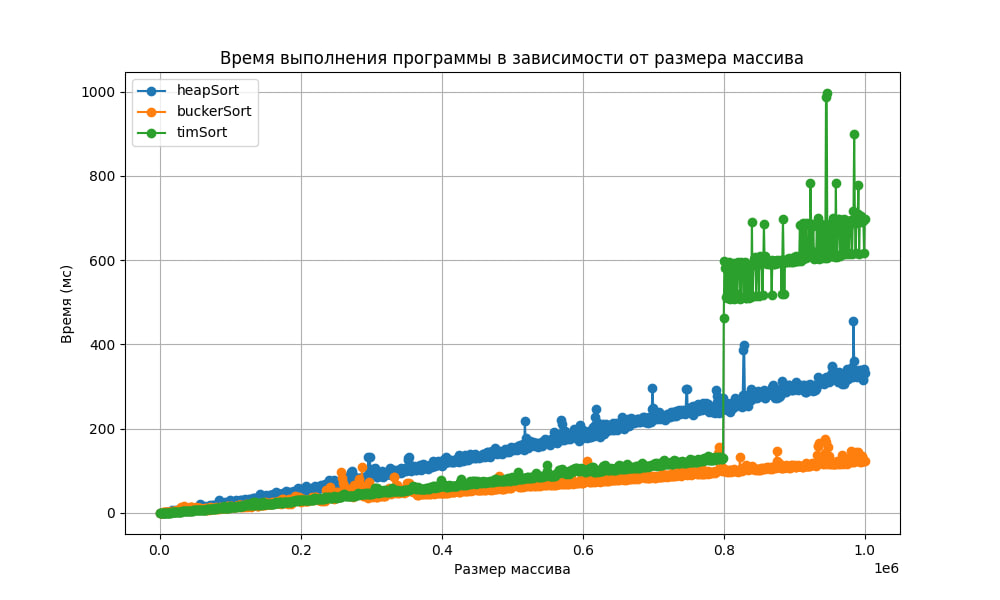


Рис.1 – график зависимости времени выполнения сортировок от размера массива

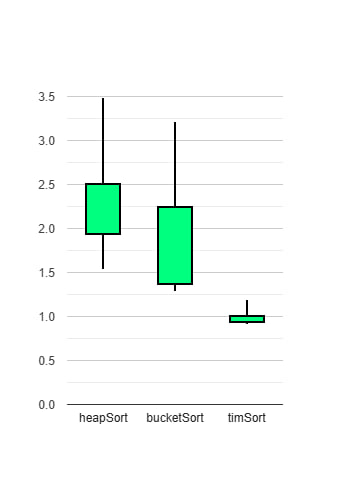


Рис.2 – boxplot графики для сортировок массивов с числом элементов 1е4

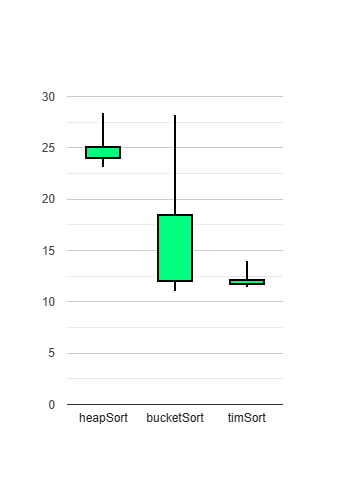


Рис.3 - boxplot графики для сортировок массивов с числом элементов 1е5

**Heap Sort**

**Асимптотика:** O(nlog n) для худшего и среднего случая.

**На практике:** линия для Heap Sort демонстрирует достаточно стабильный рост времени выполнения по мере увеличения размера массива. Это соответствует его асимптотической сложности, которая растет логарифмически относительно размера данных.

**Особенности:** относительно высокая стабильность времени выполнения объясняется тем, что Heap Sort не зависит от структуры входных данных. Однако из-за большего числа операций перемещения элементов, он может быть медленнее других алгоритмов для частично отсортированных данных.

**Bucket Sort**

**Асимптотика:** O(n+k), где k — количество блоков, и O(n^2) в худшем случае (если все элементы попадают в один блок).

**На практике:** Bucket Sort показывает наиболее эффективное время выполнения среди трёх алгоритмов для большинства случаев, что соответствует его линейной асимптотике O(n) при равномерном распределении элементов.

**Особенности:** Алгоритм работает быстро, если данные равномерно распределены. Небольшие выбросы на графике возникают из-за перераспределения элементов по блокам, когда данные оказываются сгруппированы неравномерно.

**Tim Sort**

**Асимптотика:** O(nlog n) в худшем случае и O(n) при частично отсортированных данных.

**На практике:** Tim Sort показывает более высокую производительность при небольших размерах массивов и частично отсортированных данных. Однако выбросы на графике (в районе больших размеров массива) могут быть связаны с особенностями работы алгоритма слияния и дополнительными затратами памяти при объединении ранов.

**Особенности:** Выбросы указывают на наличие больших ранов или неравномерное распределение данных, что может вызвать задержки при выполнении процедуры слияния.

**Выбросы и их причины**

**Heap Sort:** Выбросы могут возникать из-за неэффективного использования кучи на больших данных, где возникают дополнительные накладные расходы на перестройку поддеревьев.

**Bucket Sort:** Небольшие выбросы могут наблюдаться, если элементы распределяются неравномерно и значительная часть элементов попадает в один блок, что приводит к ухудшению временной сложности.

**Tim Sort:** Значительные выбросы связаны с большими затратами на слияние ранов или недостаточной адаптивностью алгоритма при сильной неоднородности данных.

**Применение алгоритмов**

**Heap Sort:**

Подходит для ситуаций, где важна **стабильная производительность** независимо от структуры данных.

Эффективен для **ограниченной памяти**, поскольку не требует дополнительного пространства O(1).

Используется в системах реального времени и для работы с большими массивами данных на диске (например, внешняя сортировка).

**Bucket Sort:**

Эффективен при **равномерном распределении данных** и небольшом диапазоне значений.

Подходит для задач, где требуется **почти линейная сложность**. Например, сортировка чисел с плавающей точкой или больших массивов с небольшим диапазоном значений.

Неэффективен при неравномерном распределении и для больших данных, из-за высокой пространственной сложности.

**Tim Sort:**

Идеален для частично отсортированных данных или данных с локальной структурой.

Широко применяется в **прикладном программировании**, например, в Python (sorted()) и Java, где алгоритм адаптируется под структуру данных.

Подходит для реальных приложений, где производительность критична, а входные данные могут иметь разную природу.

5. Заключение

В ходе работы были реализованы три алгоритма сортировки: Heap Sort, Bucket Sort и Tim Sort.

Цель работы была достигнута: выполнено сравнение производительности алгоритмов на различных массивах.

Полученные результаты подтвердили теоретические оценки сложности.

Tim Sort оказался наиболее эффективным по времени выполнения на всех тестах.

ПРИЛОЖЕНИЕ А

Листинг кода файла heap\_sort.cpp

#include <iostream>  
using namespace std;  
  
// Функция для упорядочивания поддерева в структуре бинарной кучи  
void adjustHeap(int array[], int size, int root) {  
 int maxIndex = root; // Переменная `maxIndex`: 4 байта  
 int leftChild = 2 \* root + 1; // Переменная `leftChild`: 4 байта  
 int rightChild = 2 \* root + 2; // Переменная `rightChild`: 4 байта  
  
 // Проверяем, больше ли левый потомок текущего узла  
 if (leftChild < size && array[leftChild] > array[maxIndex]) {  
 maxIndex = leftChild;  
 }  
  
 // Проверяем, больше ли правый потомок текущего узла  
 if (rightChild < size && array[rightChild] > array[maxIndex]) {  
 maxIndex = rightChild;  
 }  
  
 // Если наибольший элемент не является корнем, меняем их местами  
 if (maxIndex != root) {  
 swap(array[root], array[maxIndex]); // Операция swap: O(1)  
  
 // Рекурсивно корректируем затронутое поддерево  
 adjustHeap(array, size, maxIndex); // Рекурсия: O(log n)  
 }  
 // Пространственная сложность: O(1) дополнительных переменных  
}  
  
// Функция для выполнения пирамидальной сортировки  
void performHeapSort(int array[], int size) {  
 // Формируем структуру кучи  
 for (int i = size / 2 - 1; i >= 0; i--) {  
 adjustHeap(array, size, i); // Коррекция кучи для каждого поддерева: O(log n)  
 }  
 // Временная сложность: O(n log n) на построение кучи  
 // Пространственная сложность: O(1)  
  
 // Удаляем элементы из кучи по одному  
 for (int i = size - 1; i > 0; i--) {  
 // Перемещаем текущий корневой элемент в конец  
 swap(array[0], array[i]); // O(1)  
  
 // Корректируем оставшуюся часть кучи  
 adjustHeap(array, i, 0); // O(log n)  
 }  
 // Временная сложность: O(n log n)  
 // Пространственная сложность: O(1)  
}  
  
bool areArraysEqual(int \*array1, int \*array2, int size1, int size2) {  
 // Проверка, равны ли размеры массивов  
 if (size1 != size2) {  
 return false; // Сравнение размеров: O(1)  
 }  
  
 // Проверка элементов массивов на равенство  
 for (int i = 0; i < size1; ++i) { // Цикл: O(n)  
 if (array1[i] != array2[i]) {  
 return false;  
 }  
 }  
  
 return true; // Пространственная сложность: O(1)  
}  
  
// Главная функция для тестирования сортировки  
int main() {  
 int test\_passed = 0; // Переменная `test\_passed`: 4 байта  
 int test\_failed = 0; // Переменная `test\_failed`: 4 байта  
  
 // Тестовые данные  
 int data1[] = {12, 11, 13, 5, 6, 7}; // Массив: 6 элементов \* 4 байта = 24 байта  
 int data1Size = sizeof(data1) / sizeof(data1[0]); // Размер массива: 4 байта  
 int expected1[] = {5, 6, 7, 11, 12, 13}; // Массив: 6 элементов \* 4 байта = 24 байта  
  
 int data2[] = {}; // Пустой массив: 0 байт  
 int data2Size = sizeof(data2) / sizeof(data2[0]); // Размер массива: 4 байта  
 int expected2[] = {}; // Пустой массив: 0 байт  
  
 int data3[] = {0}; // Массив из одного элемента: 4 байта  
 int data3Size = sizeof(data3) / sizeof(data3[0]); // Размер массива: 4 байта  
 int expected3[] = {0}; // Массив из одного элемента: 4 байта  
  
 // Выполнение сортировки  
 performHeapSort(data1, data1Size); // O(n log n)  
 performHeapSort(data2, data2Size); // O(n log n)  
 performHeapSort(data3, data3Size); // O(n log n)  
  
 // Проверка тестов  
 if (areArraysEqual(data1, expected1, data1Size, data1Size)) {  
 ++test\_passed;  
 cout << "[PASSED] Test 1" << endl;  
 } else {  
 ++test\_failed;  
 cout << "[NOT PASSED] Test 1" << endl;  
 }  
  
 if (areArraysEqual(data2, expected2, data2Size, data2Size)) {  
 ++test\_passed;  
 cout << "[PASSED] Test 2" << endl;  
 } else {  
 ++test\_failed;  
 cout << "[NOT PASSED] Test 2" << endl;  
 }  
  
 if (areArraysEqual(data3, expected3, data3Size, data3Size)) {  
 ++test\_passed;  
 cout << "[PASSED] Test 3" << endl;  
 } else {  
 ++test\_failed;  
 cout << "[NOT PASSED] Test 3" << endl;  
 }  
  
 // Вывод результатов  
 cout << "Tests passed: " << test\_passed << endl; // O(1)  
 cout << "Tests failed: " << test\_failed << endl; // O(1)  
  
 return 0; // Пространственная сложность: O(1)  
}

Листинг кода файла bucket\_sort.cpp

#include <iostream>  
#include <vector>  
using namespace std;  
  
// Функция для сортировки пузырьком  
void bubbleSort(vector<float> &v) {  
 int n = v.size(); // Переменная `n`: 4 байта  
 for (int i = 0; i < n - 1; i++) { // Внешний цикл: O(n)  
 for (int j = 0; j < n - i - 1; j++) { // Внутренний цикл: O(n - i)  
 if (v[j] > v[j + 1])  
 swap(v[j], v[j + 1]); // Сравнение и обмен: O(1)  
 }  
 }  
 // Временная сложность: O(n^2)  
 // Пространственная сложность: O(1), так как сортировка выполняется на месте  
}  
  
// Функция для выполнения блочной сортировки  
void bucketSort(float arr[], int n) {  
 vector<float> b[n];  
 // `b`: массив векторов, каждый вектор имеет размер O(k), где k — среднее количество элементов в блоке  
 // Пространственная сложность: O(n), так как в худшем случае все элементы могут попасть в один блок  
  
 // Распределение элементов по блокам  
 for (int i = 0; i < n; i++) { // Цикл: O(n)  
 int bi = n \* arr[i]; // Вычисление индекса блока: O(1)  
 b[bi].push\_back(arr[i]); // Добавление элемента в блок: O(1) амортизировано  
 }  
  
 // Сортировка каждого блока  
 for (int i = 0; i < n; i++) { // Цикл: O(n)  
 bubbleSort(b[i]); // Сортировка блока: O(k^2), где k — количество элементов в блоке  
 }  
 // В худшем случае суммарное время сортировки блоков: O(n^2)  
  
 // Сборка отсортированных элементов обратно в массив  
 int index = 0; // Переменная `index`: 4 байта  
 for (int i = 0; i < n; i++) { // Цикл: O(n)  
 for (int j = 0; j < b[i].size(); j++) { // Итерация по каждому блоку: O(k)  
 arr[index++] = b[i][j]; // Копирование элемента: O(1)  
 }  
 }  
 // Итоговая временная сложность: O(n^2) в худшем случае  
 // Пространственная сложность: O(n)  
}  
  
bool areArraysEqual(float \*array1, float \*array2, int size1, int size2) {  
 // Проверка, равны ли размеры массивов  
 if (size1 != size2) { // Сравнение размеров: O(1)  
 return false;  
 }  
  
 // Проверка элементов массивов на равенство  
 for (int i = 0; i < size1; ++i) { // Цикл: O(n)  
 if (array1[i] != array2[i]) {  
 return false;  
 }  
 }  
 return true; // Пространственная сложность: O(1)  
}  
  
// Главная функция для тестирования сортировки  
int main() {  
 int test\_passed = 0; // Переменная `test\_passed`: 4 байта  
 int test\_failed = 0; // Переменная `test\_failed`: 4 байта  
  
 // Тестовые данные  
 float data1[] = {0.11, 0.05, 0.15, 0.25, 0.04, 0.99}; // Массив: 6 элементов \* 4 байта = 24 байта  
 int data1Size = sizeof(data1) / sizeof(data1[0]); // Размер массива: 4 байта  
 float expected1[] = {0.04, 0.05, 0.11, 0.15, 0.25, 0.99}; // Массив: 6 элементов \* 4 байта = 24 байта  
  
 float data2[] = {}; // Пустой массив: 0 байт  
 int data2Size = sizeof(data2) / sizeof(data2[0]); // Размер массива: 4 байта  
 float expected2[] = {}; // Пустой массив: 0 байт  
  
 float data3[] = {0.01}; // Массив из одного элемента: 4 байта  
 int data3Size = sizeof(data3) / sizeof(data3[0]); // Размер массива: 4 байта  
 float expected3[] = {0.01}; // Массив из одного элемента: 4 байта  
  
 // Выполнение сортировки  
 bucketSort(data1, data1Size); // O(n^2) в худшем случае  
 bucketSort(data2, data2Size); // O(n^2) в худшем случае  
 bucketSort(data3, data3Size); // O(n^2) в худшем случае  
  
 // Проверка тестов  
 if (areArraysEqual(data1, expected1, data1Size, data1Size)) {  
 ++test\_passed;  
 cout << "[PASSED] Test 1" << endl;  
 } else {  
 ++test\_failed;  
 cout << "[NOT PASSED] Test 1" << endl;  
 }  
  
 if (areArraysEqual(data2, expected2, data2Size, data2Size)) {  
 ++test\_passed;  
 cout << "[PASSED] Test 2" << endl;  
 } else {  
 ++test\_failed;  
 cout << "[NOT PASSED] Test 2" << endl;  
 }  
  
 if (areArraysEqual(data3, expected3, data3Size, data3Size)) {  
 ++test\_passed;  
 cout << "[PASSED] Test 3" << endl;  
 } else {  
 ++test\_failed;  
 cout << "[NOT PASSED] Test 3" << endl;  
 }  
  
 // Вывод результатов  
 cout << "Tests passed: " << test\_passed << endl; // O(1)  
 cout << "Tests failed: " << test\_failed << endl; // O(1)  
  
 return 0; // Пространственная сложность: O(1)  
}

Листинг кода файла tim\_sort.cpp

#include <iostream>  
#include <algorithm> // Для функции min  
using namespace std;  
  
const int RUN = 32; // Размер подмассива для Insertion Sort  
  
// Функция сортировки вставками  
void insertionSort(int arr[], int left, int right) {  
 for (int i = left + 1; i <= right; i++) {  
 int temp = arr[i]; // Временная переменная для вставки: 4 байта  
 int j = i - 1;  
 while (j >= left && arr[j] > temp) { // O(n) в худшем случае  
 arr[j + 1] = arr[j]; // Сдвиг элементов: O(1)  
 j--;  
 }  
 arr[j + 1] = temp; // Вставка элемента: O(1)  
 }  
 // Временная сложность: O(n^2) для подмассива длины n  
 // Пространственная сложность: O(1), так как сортировка выполняется на месте  
}  
  
// Функция слияния двух отсортированных частей массива  
void merge(int arr[], int l, int m, int r) {  
 int len1 = m - l + 1, len2 = r - m;  
 int left[len1], right[len2];  
 // Временные массивы для слияния: O(len1 + len2)  
  
 // Копирование элементов в временные массивы  
 for (int i = 0; i < len1; i++)  
 left[i] = arr[l + i]; // O(len1)  
 for (int i = 0; i < len2; i++)  
 right[i] = arr[m + 1 + i]; // O(len2)  
  
 // Слияние двух массивов обратно в arr[l..r]  
 int i = 0, j = 0, k = l;  
 while (i < len1 && j < len2) { // O(len1 + len2)  
 if (left[i] <= right[j]) {  
 arr[k++] = left[i++];  
 } else {  
 arr[k++] = right[j++];  
 }  
 }  
  
 // Копирование оставшихся элементов  
 while (i < len1)  
 arr[k++] = left[i++]; // O(len1)  
 while (j < len2)  
 arr[k++] = right[j++]; // O(len2)  
  
 // Временная сложность: O(len1 + len2)  
 // Пространственная сложность: O(len1 + len2) для временных массивов  
}  
  
// TimSort  
void timSort(int arr[], int n) {  
 // Сортировка подмассивов размером RUN с помощью сортировки вставками  
 for (int i = 0; i < n; i += RUN)  
 insertionSort(arr, i, min((i + RUN - 1), (n - 1))); // O(RUN^2 \* (n / RUN))  
  
 // Слияние подмассивов  
 for (int size = RUN; size < n; size = 2 \* size) { // Логарифмический рост размера подмассивов: O(log(n / RUN))  
 for (int left = 0; left < n; left += 2 \* size) { // Количество слияний: O(n / size)  
 int mid = left + size - 1;  
 int right = min((left + 2 \* size - 1), (n - 1));  
 if (mid < right)  
 merge(arr, left, mid, right); // O(size)  
 }  
 }  
 // Временная сложность:  
 // - O(n log n) для слияния  
 // - O(RUN^2 \* (n / RUN)) ≈ O(n) для сортировки вставками  
 // Итоговая временная сложность: O(n log n)  
 // Пространственная сложность: O(n) из-за временных массивов для слияния  
}  
  
// Проверка на равенство массивов  
bool areArraysEqual(int \*array1, int \*array2, int size1, int size2) {  
 if (size1 != size2) return false; // O(1)  
 for (int i = 0; i < size1; ++i) {  
 if (array1[i] != array2[i]) return false; // O(n)  
 }  
 return true; // Пространственная сложность: O(1)  
}  
  
// Главная функция  
int main() {  
 int test\_passed = 0; // 4 байта  
 int test\_failed = 0; // 4 байта  
  
 // Тестовые данные  
 int data1[] = {12, 11, 13, 5, 6, 7}; // 6 элементов: 24 байта  
 int data1Size = sizeof(data1) / sizeof(data1[0]); // 4 байта  
 int expected1[] = {5, 6, 7, 11, 12, 13}; // 6 элементов: 24 байта  
  
 int data2[] = {}; // Пустой массив: 0 байт  
 int data2Size = sizeof(data2) / sizeof(data2[0]); // 4 байта  
 int expected2[] = {}; // Пустой массив: 0 байт  
  
 int data3[] = {0}; // 1 элемент: 4 байта  
 int data3Size = sizeof(data3) / sizeof(data3[0]); // 4 байта  
 int expected3[] = {0}; // 1 элемент: 4 байта  
  
 // Выполнение сортировки  
 timSort(data1, data1Size); // O(n log n)  
 timSort(data2, data2Size); // O(1)  
 timSort(data3, data3Size); // O(1)  
  
 // Проверка тестов  
 if (areArraysEqual(data1, expected1, data1Size, data1Size)) {  
 ++test\_passed;  
 cout << "[PASSED] Test 1" << endl;  
 } else {  
 ++test\_failed;  
 cout << "[NOT PASSED] Test 1" << endl;  
 }  
 if (areArraysEqual(data2, expected2, data2Size, data2Size)) {  
 ++test\_passed;  
 cout << "[PASSED] Test 2" << endl;  
 } else {  
 ++test\_failed;  
 cout << "[NOT PASSED] Test 2" << endl;  
 }  
 if (areArraysEqual(data3, expected3, data3Size, data3Size)) {  
 ++test\_passed;  
 cout << "[PASSED] Test 3" << endl;  
 } else {  
 ++test\_failed;  
 cout << "[NOT PASSED] Test 3" << endl;  
 }  
  
 // Вывод результатов  
 cout << "Tests passed: " << test\_passed << endl; // O(1)  
 cout << "Tests failed: " << test\_failed << endl; // O(1)  
  
 return 0;  
}