ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 5

«Сортировка»

Выполнил работу

Дубинина Диана

Академическая группа C3100

Принято

Ментор Вершинин Владислав

Санкт-Петербург

2024

1. **Введение**

Целью данной лабораторной работы является умение работать с различными способами реализации алгоритмов сортировки

Задачи: Нахождение трех алгоритмов сортировки, подходящих под различные критерии, указанные в таблице, но отличных от тех, что есть в ТЗ; реализация всех трех на языке программирования C++ и их анализ

1. **Теоретическая подготовка**

Для выполнения лабораторной работы по теме "Алгоритмы сортировки на C++" важно изучить их основные виды, сложность и используемые структуры данных. Сортировка — это процесс упорядочивания элементов массива в заданном порядке. Различают алгоритмы сортировки по сложности, способу выполнения и устойчивости. Существует множество различных алгоритмов сортировки, но в этой работе будут реализованы «Шейкерная сортировка», «Пирамидальная сортировка» и «Блочная сортировка». Эффективность алгоритмов оценивается по времени выполнения и использованию памяти. Простые алгоритмы подходят для небольших массивов, тогда как более сложные — для больших данных. Реализация на C++ требует использования циклов, условных конструкций и функций, что позволяет гибко адаптировать алгоритмы под различные задачи. Знание особенностей каждого алгоритма позволит эффективно применять их на практике в зависимости от задач и ограничений. Также важно владеть навыками оценки времени выполнения программы и анализа вычислительной сложности алгоритма.

1. **Реализация**

В процессе работы использовалась стандартная библиотека C++: iostream - для вывода сообщений на экран; vector - для хранения элементов, так как вектор позволяет динамически изменять размер и эффективно работать с данными; cassert - для реализации проверок с помощью assert, которые гарантируют, что массивы после сортировки совпадают с ожидаемыми результатами; algorithm - для использования функции swap, которая меняет элементы местами.

На первом этапе был выбран алгоритм шейкерной сортировки, являющийся модификацией пузырьковой сортировки. Основная идея алгоритма заключается в том, чтобы перемещать элементы в двух направлениях: сначала слева направо, а затем справа налево. Это позволяет "всплывать" наибольшим элементам в конец массива и "опускать" наименьшим в начало массива.

В качестве входных данных использовался вектор целых чисел, что позволяло гибко работать с массивами разного размера.

Алгоритм был реализован с использованием двух основных циклов: Первый цикл проходит массив слева направо, сравнивая соседние элементы и переставляя их, если они идут в неправильном порядке. Второй цикл идёт справа налево, аналогично перемещая элементы, но в другую сторону.

Использование переменных left и right помогает отслеживать границы неотсортированной части массива. Если на каком-то из этапов не было перестановок (флаг swapped остался равным false), сортировка завершается досрочно.

Вот как выглядит ключевая часть кода:

for (int i = left; i < right; ++i) {

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

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

swapped = true;

}

}

--right;

for (int i = right; i > left; --i) {

if (arr[i] < arr[i - 1]) {

swap(arr[i], arr[i - 1]);

swapped = true;

}

}

++left;

Пирамидальная сортировка — это алгоритм, основанный на создании кучи (heap). В данном случае искусственно добавлено ухудшение производительности, чтобы продемонстрировать, как дополнительные операции влияют на эффективность.

Функция inefficientHeapify **-** Строит кучу, начиная с указанного корня. Проверяет, является ли текущий элемент (i) наибольшим среди своих потомков. Если нет, то обменивает значения и рекурсивно вызывает себя. Добавлено искусственное ухудшение: после каждой перестановки пересоздается вся куча.

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

int largest = i; // Корень

int left = 2 \* i + 1;

int right = 2 \* i + 2;

// Находим наибольший элемент среди корня, левого и правого потомков

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

largest = left;

}

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

largest = right;

}

// Если наибольший элемент не корень

if (largest != i) {

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

// Искусственно создаём неэффективность: пересоздаём кучу целиком

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

inefficientHeapify(arr, n, j);

}

}

}

Основная функция heapSort: строит кучу для всего массива. Последовательно перемещает максимальный элемент (корень) в конец массива и перестраивает уменьшенную кучу.

void heapSort(vector<int>& arr) {

int n = arr.size();

// Построение начальной кучи

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

inefficientHeapify(arr, n, i);

}

// Извлечение элементов из кучи

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

// Перемещаем текущий корень в конец

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

// Восстанавливаем кучу на уменьшенном массиве

inefficientHeapify(arr, i, 0);

}

}

В данном случае включает искусственное ухудшение производительности. Обычно алгоритм строит двоичную кучу, где каждый узел больше или равен своим потомкам, а затем извлекает элементы по одному, начиная с корня (максимального элемента). Однако в функции inefficientHeapify после каждой перестановки кучи выполняется полная перестройка, что значительно увеличивает количество операций. Это ухудшает асимптотику с O(NlogN) до примерно O(N^2). Такая модификация демонстрирует, как избыточные вычисления могут существенно замедлить алгоритм.

Блочная сортировка делит элементы на несколько корзин (buckets), сортирует их, а затем объединяет. Реализация: Находятся минимальное (minValue) и максимальное (maxValue) значения. Рассчитывается количество корзин (bucketCount), исходя из размера корзины (bucketSize). Элементы массива распределяются по корзинам на основе их значений. Каждая корзина сортируется (с помощью std::sort), и её элементы добавляются обратно в основной массив.

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

if (arr.empty()) {

return;

}

// Находим минимальное и максимальное значение в массиве

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

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

// Количество корзин

int bucketCount = (maxValue - minValue) / bucketSize + 1;

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

// Распределяем элементы массива по корзинам

for (int num : arr) {

int bucketIndex = (num - minValue) / bucketSize;

buckets[bucketIndex].push\_back(num);

}

// Сортируем каждую корзину и объединяем результат

arr.clear();

for (vector<int>& bucket : buckets) {

sort(bucket.begin(), bucket.end()); // Внутренняя сортировка корзины

arr.insert(arr.end(), bucket.begin(), bucket.end());

}

}

После того как алгоритмы были написаны, внимание было уделено тестированию них на различных наборах данных, чтобы удостовериться в их правильности и эффективности в разных случаях, для проверки результатов использовалась функция assert, которая сравнивает отсортированный массив с ожидаемым результатом. Пример кода для тестирования лучшего случая:

void testBestCase() {

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

vector<int> expected = arr; // Массив уже отсортирован

shakerSort(arr);

heapSort(arr);

bucketSort(arr, 2);

assert(arr == expected && "Test Best Case Failed!");

}

После реализации алгоритма и тестов был запущен процесс тестирования. Все тесты прошли успешно, и программа вывела сообщение о том, что все тесты были пройдены:

cout << "Все тесты успешно пройдены!" << endl;

1. **Экспериментальная часть**

Подсчитаем память, используемую в данном коде.

**Переменные**: В шейкерной сортировке используются три основные переменные: left (4 байта), right (4 байта) и swapped (1 байт). Таким образом, общий объем памяти, необходимый для этих переменных, составляет 9 байт. Эти переменные находятся на стеке, и их размер остается постоянным независимо от размера входного массива.

**Динамический массив vector<int>**: Каждый объект vector<int> включает указатель на данные (8 байт), текущий размер массива (4 байта) и емкость (4 байта). Для массива, содержащего N элементов, дополнительно выделяется N⋅4 байта памяти, так как каждый элемент занимает 4 байта. Таким образом, общий объем памяти, занимаемый массивом, равен 16+N⋅4 байт.

**Внутренние функции:** **shakerSort** - Функция использует переменные left, right и swapped, а также два цикла. Память на стеке для этой функции составляет 9 байт для переменных плюс накладные расходы вызова функции (~16 байт). Общий объем памяти, используемой на стеке, составляет примерно 25 байт. **inefficientHeapify (используется в heapSort)** - Функция использует переменные largest (4 байта), left (4 байта) и right (4 байта). Каждый вызов функции дополнительно занимает ~16 байт на стеке в виде накладных расходов. Общий объем памяти для одной итерации функции составляет около 28 байт. **bucketSort -** Для блочной сортировки создается массив корзин vector<vector<int>> buckets. Каждая корзина (объект vector<int>) включает указатель, размер и емкость (16 байт), а также элементы. Если корзин K, а входной массив содержит N элементов, то в среднем каждая корзина содержит N/K элементов. Таким образом, объем памяти для всех корзин можно оценить как K⋅16+N⋅4.

**Тесты**: Каждый тест создает массивы, размеры которых зависят от случая. Например, тест для 10 элементов требует 16+10⋅4=56 байт памяти.

**Функция measureTime:** Функция measureTime работает с массивами размером от 1000 до 10000 элементов. Для массива размера N, память, занимаемая массивом, составляет 16+N⋅4 байт.

* Для массива из 1000 элементов: 16+1000⋅4= 4016 байт.
* Для массива из 10000 элементов: 16+10000⋅4=40016 байт.

**Итог:** Шейкерная сортировка требует постоянного объема памяти на стеке (~25 байт) плюс память для массива: 16+N⋅4. Итоговая пространственная сложность составляет O(N).Пирамидальная сортировка также использует стек (~28 байт на итерацию) и массив размером N. Пространственная сложность составляет O(N).Блочная сортировка создает дополнительную структуру для корзин. Объем используемой памяти зависит от количества корзин K и количества элементов N: K⋅16+N⋅4. Пространственная сложность составляет O(N⋅K) где K≪N.

Таким образом, все алгоритмы укладываются в заданные ограничения по пространственной сложности.

**Асимптотика для алгоритма №1:** Шейкерная сортировка, или двунаправленная пузырьковая сортировка, имеет следующую асимптотику.

Во временной сложности она демонстрирует разные результаты в зависимости от ситуации. В худшем случае, когда массив полностью отсортирован в обратном порядке, на каждом проходе требуется N−1, N−2, N−3 и так далее сравнений и обменов, что суммарно даёт (N×(N−1))/2​. Это соответствует сложности O(N^2). Количество обменов также пропорционально количеству сравнений, что подтверждает временную сложность O(N^2).

В лучшем случае, когда массив уже отсортирован, алгоритм завершает работу после одного прохода, так как флаг swapped остаётся равным false. В этом случае потребуется только N−1 сравнений, что даёт временную сложность O(N).

В среднем случае, когда массив имеет случайный порядок элементов, в среднем выполняется половина всех возможных сравнений и обменов. Это примерно (N×(N−1))/4​, что также соответствует сложности O(N^2).

Пространственная сложность алгоритма равна O(1), так как алгоритм работает на месте, не используя дополнительных структур данных. Он использует лишь три временные переменные: left, right для границ массива (по 4 байта каждая) и swapped для проверки наличия перестановок (1 байт).

Итог: Худший случай O(N^2); средний случай O(N^2); лучший случай O(N); пространственная сложность O(1).

Таким образом, шейкерная сортировка наиболее эффективна для небольших массивов или данных, которые уже частично отсортированы. Однако для больших массивов её временная сложность делает её менее предпочтительной по сравнению с более продвинутыми алгоритмами, такими как быстрая сортировка или сортировка слиянием.

**Шейкерная сортировка (shakerSort)**: В лучшем случае (массив уже отсортирован): Каждый проход по массиву проверяет элементы, не производя перестановок. За O(N) проверок сортировка завершается. В худшем случае (массив отсортирован в обратном порядке): Каждый элемент перемещается на своё место, что требует N+(N−1)+(N−2)+…+1= O(N^2) операций. В среднем случае: Ожидаемое количество перестановок и сравнений также составляет порядка O(N^2). Пространственная сложность: Алгоритм использует фиксированное количество переменных на стеке, не создавая дополнительных структур данных, то есть сложность будет O(1).

**Пирамидальная сортировка (heapSort)**: Временная сложность: Построение кучи (инициализация) - Выполняется для каждого узла O(log N) операций, всего O(N) узлов, т.е сложность равна O(NlogN). Извлечение элементов - Для каждого из N элементов требуется O(logN) операций для восстановления кучи, итог - O(NlogN). Однако в данном случае используется неэффективная куча, где пересоздание на каждом шаге увеличивает затраты до O(N^2). Пространственная сложность: Используется один массив для хранения элементов. Память на стеке — фиксированная, сложность равна O(1).

**Блочная сортировка (bucketSort):** Временная сложность - Распределение элементов по корзинам:Для каждого из N элементов определяется корзина, это O(N).Внутренняя сортировка корзин:Для каждой из K корзин, сортировка занимает в среднем O((N/K)log(N/K)) времени. Для всех корзин: O(K⋅(N/K)log(N/K))=O(Nlog(N/K))**.** Итоговая сложность - O(Nlog(N/K)), где K≪N.Пространственная сложность:Создается K корзин, каждая из которых в среднем содержит N/K элементов, итог - O(N+K).

Таблица №1 – асимптотический анализ алгоритмов

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Алгоритм | Лучший случай | Средний случай | Худший случай | Пространственная сложность |
| №1(шейкерная) | O(N) | O(N^2) | O(N^2) | O(1) |
| №2(пирамидальная) | O(N^2) | O(N^2) | O(N^2) | O(1) |
| №3(блочная) | O(N+K) | O(NlogN/K) | O(N^2) | O(N+K) |

Согласно требованиям моего варианта, на вход к моему алгоритму подаётся от 1000 до 1е6 элементов. Было выполнено нахождение зависимости времени выполнения от размера массива для Max 10000 эл-тов. Для тестирования алгоритма была собрана статистика, приведенная в таблице №2.

Таблица №2 - зависимости времени от числа элементов

|  |  |  |  |
| --- | --- | --- | --- |
| Размер | Алгоритм 1 | Алгоритм 2 | Алгоритм 3 |
| 1000 | 0,005307 | 0,026368 | 0,000264 |
| 2000 | 0,022014 | 0,115799 | 0,000378 |
| 3000 | 0,046519 | 0,292087 | 0,000551 |
| 4000 | 0,085697 | 0,540831 | 0,000641 |
| 5000 | 0,140386 | 0,852568 | 0,000748 |
| 6000 | 0,196398 | 1,24487 | 0,000901 |
| 7000 | 0,260285 | 1,74117 | 0,001043 |
| 8000 | 0,335236 | 2,18916 | 0,001198 |
| 9000 | 0,418175 | 2,76422 | 0,001307 |
| 10000 | 0,522535 | 3,4041 | 0,001425 |

График представляющий визуально удобный формат данных из таблицы №2 представлен на изображении №1.

Изображение выглядит как текст, линия, График, диаграмма

Автоматически созданное описание

Изображение №1 - График работы алгоритмов

Заметим, что при относительно небольших значениях N время работы алгоритмов практически равно нулю. Это значит, что при небольших входных наборах алгоритмы будет выполняться очень быстро, но при увеличении количества входных данных алгоритмы работает медленнее. Однако также сравнив графики работы алгоритмов между собой, мы заметим, что наиболее эффективным является №3 (блочный) и наименее - №2 (пирамидальный).

Изображение выглядит как текст, снимок экрана, число, Шрифт

Автоматически созданное описание

Изображение №2 – график box plot для массивов 1е5

Боксплот, представляющий визальную диаграмму для «среднего» времени, которое требуется для выполнения сортировки.

1. **Заключение**

В ходе работы были реализованы шейкерная, пирамидальная (с искусственным ухудшением), и блочная сортировки, а их теоретическая асимптотика сопоставлена с практическими результатами.

Шейкерная сортировка показала эффективность на частично отсортированных массивах, подтверждая асимптотику O(N) в лучшем случае. Однако на случайных и обратных данных время выполнения достигало O(N^2), что делает её непригодной для больших массивов. Пирамидальная сортировка из-за искусственного ухудшения продемонстрировала нестабильную производительность и значительное отклонение от стандартного поведения. Она неэффективна даже для средних размеров массивов. Блочная сортировка подтвердила свою асимптотику O(N\log(N/K)), показывая высокую производительность на равномерно распределённых данных. Однако при неравномерном распределении возникали выбросы из-за перегруженных корзин.

Шейкерная сортировка подходит для небольших массивов с частичной упорядоченностью, пирамидальная в текущей реализации не рекомендуется, а блочная эффективна для больших массивов с равномерным распределением данных. Выбор алгоритма зависит от структуры входных данных и требований к скорости.

ПРИЛОЖЕНИЕ А

Листинг кода, реализованного в лабораторной работе

#include <iostream>

#include <vector>

#include <cassert>

#include <algorithm> // Для std::is\_sorted

using namespace std;

// Реализация шейкерной сортировки

void shakerSort(vector<int>& arr) {

int left = 0; // Левая граница неотсортированной части массива

int right = arr.size() - 1; // Правая граница неотсортированной части массива

bool swapped; // Флаг для проверки наличия перестановок

do {

swapped = false;

// Проход слева направо: сдвигаем наибольший элемент в конец массива

for (int i = left; i < right; ++i) {

if (arr[i] > arr[i + 1]) { // Если текущий элемент больше следующего

swap(arr[i], arr[i + 1]); // Меняем их местами

swapped = true; // Отмечаем, что была перестановка

}

}

--right; // Сужаем правую границу, так как последний элемент на своём месте

// Проход справа налево: сдвигаем наименьший элемент в начало массива

for (int i = right; i > left; --i) {

if (arr[i] < arr[i - 1]) { // Если текущий элемент меньше предыдущего

swap(arr[i], arr[i - 1]); // Меняем их местами

swapped = true; // Отмечаем, что была перестановка

}

}

++left; // Сужаем левую границу, так как первый элемент на своём месте

} while (swapped); // Продолжаем, пока происходят перестановки

}

// Функция для создания кучи с искусственным ухудшением производительности

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

int largest = i; // Корень

int left = 2 \* i + 1;

int right = 2 \* i + 2;

// Находим наибольший элемент среди корня, левого и правого потомков

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

largest = left;

}

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

largest = right;

}

// Если наибольший элемент не корень

if (largest != i) {

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

// Искусственно создаём неэффективность: пересоздаём кучу целиком

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

inefficientHeapify(arr, n, j);

}

}

}

// Основная функция пирамидальной сортировки

void heapSort(vector<int>& arr) {

int n = arr.size();

// Построение начальной кучи

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

inefficientHeapify(arr, n, i);

}

// Извлечение элементов из кучи

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

// Перемещаем текущий корень в конец

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

// Восстанавливаем кучу на уменьшенном массиве

inefficientHeapify(arr, i, 0);

}

}

// Реализация блочной сортировки

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

if (arr.empty()) {

return;

}

// Находим минимальное и максимальное значение в массиве

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

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

// Количество корзин

int bucketCount = (maxValue - minValue) / bucketSize + 1;

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

// Распределяем элементы массива по корзинам

for (int num : arr) {

int bucketIndex = (num - minValue) / bucketSize;

buckets[bucketIndex].push\_back(num);

}

// Сортируем каждую корзину и объединяем результат

arr.clear();

for (vector<int>& bucket : buckets) {

sort(bucket.begin(), bucket.end()); // Внутренняя сортировка корзины

arr.insert(arr.end(), bucket.begin(), bucket.end());

}

}

// Функция для тестирования лучшего случая

void testBestCase() {

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

vector<int> expected = arr; // Массив уже отсортирован

shakerSort(arr);

heapSort(arr);

bucketSort(arr, 2);

assert(arr == expected && "Test Best Case Failed!");

}

// Функция для тестирования среднего случая

void testAverageCase() {

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

vector<int> expected = {1, 2, 3, 4, 5, 6, 7, 8}; // Ожидаемый результат

shakerSort(arr);

heapSort(arr);

bucketSort(arr, 2);

assert(arr == expected && "Test Average Case Failed!");

}

// Функция для тестирования худшего случая

void testWorstCase() {

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

vector<int> expected = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; // Ожидаемый результат

shakerSort(arr);

heapSort(arr);

bucketSort(arr, 2);

assert(arr == expected && "Test Worst Case Failed!");

}

// Тест для пустого массива

void testEmptyArray() {

vector<int> arr = {};

vector<int> expected = {};

shakerSort(arr);

heapSort(arr);

bucketSort(arr, 2);

assert(arr == expected && "Test Empty Array Failed!");

}

// Тест для массива с одним элементом

void testSingleElement() {

vector<int> arr = {42};

vector<int> expected = {42};

shakerSort(arr);

heapSort(arr);

bucketSort(arr, 2);

assert(arr == expected && "Test Single Element Failed!");

}

// Тест для массива с одинаковыми элементами

void testAllElementsSame() {

vector<int> arr = {5, 5, 5, 5, 5};

vector<int> expected = arr; // Массив остаётся неизменным

shakerSort(arr);

heapSort(arr);

bucketSort(arr, 2);

assert(arr == expected && "Test All Elements Same Failed!");

}

// Тест для массива с отрицательными числами

void testNegativeNumbers() {

vector<int> arr = {-3, -1, -7, -4, -5, -2};

vector<int> expected = {-7, -5, -4, -3, -2, -1}; // Ожидаемый результат

shakerSort(arr);

heapSort(arr);

bucketSort(arr, 2);

assert(arr == expected && "Test Negative Numbers Failed!");

}

void measureTime() {

for (int size = 1000; size <= 10000; size += 1000) {

vector<int> arr(size);

// Заполнение массива случайными числами

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

arr[i] = rand() % 10000; // Генерация случайных чисел

}

// Засекаем время начала

clock\_t start1 = clock();

// Выполняем шейкерную сортировку

shakerSort(arr);

// Засекаем время окончания

clock\_t end1 = clock();

// Засекаем время начала

clock\_t start2 = clock();

// Выполняем пирамидальную сортировку

heapSort(arr);

// Засекаем время окончания

clock\_t end2 = clock();

// Засекаем время начала

clock\_t start3 = clock();

// Выполняем блочную сортировку

bucketSort(arr, 50);

// Засекаем время окончания

clock\_t end3 = clock();

// Вычисляем время выполнения

double duration1 = double(end1 - start1) / CLOCKS\_PER\_SEC;

double duration2 = double(end2 - start2) / CLOCKS\_PER\_SEC;

double duration3 = double(end3 - start3) / CLOCKS\_PER\_SEC;

// Выводим размер массива и время выполнения

cout << "Array size: " << size << " Time1: " << duration1 << " Time2: " << duration2 << " Time3: " << duration3 << " seconds" << endl;

}

}

// Главная функция для запуска тестов

int main() {

testBestCase();

testAverageCase();

testWorstCase();

testEmptyArray();

testSingleElement();

testAllElementsSame();

testNegativeNumbers();

measureTime();

cout << "Все тесты успешно пройдены!" << endl;

return 0;

}