ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 5

«Сортировка»

Выполнил работу

Азариан Сайед Даниал

Академическая группа № J3111

Принято

Должность, звание Фамилия Имя преподавателя

Санкт-Петербург

2024

1. Введение

Целью данной лабораторной работы является изучение и практическая реализация различных алгоритмов сортировки с разной временной сложностью. Работа позволяет глубже понять принципы и особенности алгоритмов сортировки.

В рамках этой цели решаются следующие задачи:

1. Реализовать три алгоритма сортировки с различной временной сложностью.
2. Проанализировать временную и пространственную сложность каждого из алгоритмов.
3. Сравнить результаты работы алгоритмов при различных условиях и входных данных, оценить производительность и выявить области их оптимального применения.
4. Теоретическая подготовка

Cocktail Sort — это улучшенный вариант сортировки пузырьком. Основное отличие в том, что массив сортируется в обоих направлениях. За один проход большие элементы перемещаются к концу массива, а маленькие — к началу.

* Сложность: в худшем и среднем случаях O(n^2), в лучшем случае O(n) при почти отсортированных массивах.
* Особенности: подходит для небольших или частично отсортированных массивов, не требует дополнительной памяти.

Heap Sort — использует двоичную кучу, где массив сначала преобразуется в кучу, затем на каждом шаге наибольший элемент перемещается в конец массива.

* Сложность: O(n log n) для всех случаев.
* Особенности: эффективен на больших массивах, выполняется "на месте", не является стабильным.

Bead Sort — подходит для сортировки неотрицательных целых чисел, используя принцип "гравитации". Каждый элемент представлен вертикальными "бусинами", которые "падают" вниз, формируя отсортированный массив.

* + Сложность: зависит от суммы значений в массиве O(s).
  + Особенности: применим только для неотрицательных целых чисел, требует дополнительной памяти для размещения бусин.

Реализация

В данной лабораторной работе были выполнены следующие этапы:

1. Реализация Cocktail Sort.
   1. Инициализация переменных.
   2. Проход слева направо и наоборот сортируя массив.
   3. Проверка завершения.

Повторяем процесс, пока не окажется, что на очередном проходе не было перестановок, что свидетельствует об окончательной сортировке массива

1. Реализация Heap Sort
   1. Построение кучи.

Сначала преобразуем исходный массив в двоичную кучу. Процесс начинается с последнего неполного узла и идет к корню массива. Для каждого узла вызывается функция heapify, которая проверяет его дочерние элементы и перемещает наибольший элемент вверх.

* 1. Сортировка с использованием кучи

На этом этапе самый большой элемент (корень кучи) перемещается в конец массива, а оставшаяся часть массива перестраивается, чтобы вновь получить кучу. После каждого извлечения максимального элемента вызывается heapify, чтобы восстановить структуру кучи.

* 1. Функция heapify

Этот вспомогательный метод отвечает за поддержание свойств кучи. При вызове на элементе он проверяет его дочерние узлы, перемещая элемент вниз до тех пор, пока не будут выполнены условия max-heap.

1. Реализация Bead Sort
   1. Создание структуры бусин

Инициализируется двухмерный массив, где каждая строка представляет собой "прутики", на которых располагаются бусины. Высота каждого столбика соответствует значению каждого элемента в массиве.

* 1. "Падение" бусин

После создания структуры бусин каждая "бусина" в столбике начинает "падать" вниз, подобно песчинкам в песочных часах. В результате все бусины перемещаются к нижней части прутиков.

При падении бусины оказываются на минимально возможной высоте в столбце, что позволяет быстро сформировать отсортированный массив.

* 1. Чтение отсортированного массива

После завершения падения бусин высота столбцов преобразуется обратно в значения массива, которые соответствуют отсортированному порядку.

1. Экспериментальная часть
2. Подсчет по памяти
   1. Cocktail Sort.

- Исходный массив – O(N)

- Целочисленные переменные (три) – O(1)

Итог: Общая память O(N)

* 1. Heap sort.

- Исходный массив – O(N)

- Целочисленные переменные (три) – O(1)

Итог: Общая память O(N)

* 1. Bead sort.

- Исходный массив – O(N)

- Целочисленные переменные (один) – O(N)

- Двухмерный вектор, размером n × max, где каждый элемент занимает занимает O(1) – O(n × max)

- Массив для хранения отсортированного результата – O(N)

Итог: Общая память O(N + n × max)

1. Подсчет асимптотики
   1. Cocktail sort

O(N2) в худшем и среднем случаях, так как алгоритм требует порядка n проходов, где каждый проход выполняет n операций. В лучшем случае, если массив уже отсортирован, сложность составляет O(N).

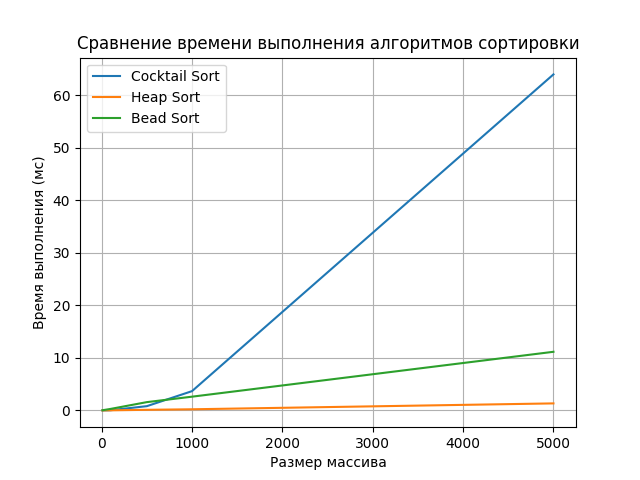
* 1. Heap sort

O(n log2n)для всех случаев (худшего, среднего и лучшего), так как процесс построения и разрушения кучи одинаков по сложности.

* 1. Bead sort

O(n × max) в худшем случае

Изображение № 1 – График работы алгоритма



Изображение №2 – box plot для 1e4 (общий график)

Изображение выглядит как текст, снимок экрана, диаграмма

Автоматически созданное описание

Изображение выглядит как текст, диаграмма, линия, План

Автоматически созданное описаниеИзображение №3 – box plot 1e4 (отдельные графики)

Изображение выглядит как текст, снимок экрана, Прямоугольник

Автоматически созданное описаниеИзображение №4 – box plot для 1e5 (общий график)

Изображение выглядит как диаграмма, текст, линия, Параллельный

Автоматически созданное описаниеИзображение №5 – box plot для 1e5 (отдельные графики)

Выводы.

1. Асимптотика и практика:

• CocktaiI Sort демонстрирует, что на практике его производительность в худших и средних случаях соответствует теоретической сложности O(N2). Алгоритм чувствителен к почти отсортированным массивам, где время работы стремится к О(N). Его реальное применение ограничено небольшими массивами или массивами с высоким уровнем упорядоченности.

• Неар Sort стабилен на практике и демонстрирует сложность log П) для всех случаев. Время выполнения алгоритма практически совпадает с теоретическим анализом благодаря детерминированной структуре кучи.

• Bead Sort показывает, что его производительность сильно зависит отвеличины элементов массива (тас). При больших значениях элементов и больших объемах данных возникают значительные затраты памяти и времени из-за инициализации и работы с двухмерной структурой бусин.

2. Выбросы.

На графиках с большими массивами могут наблюдаться выбросы, связанные с особенностями реализации (например, доступ к памяти или порядок вызовов функций).

Для Bead Sort выбросы часто связаны с ограничением на входные данные (например, большие значения тает) и высокой зависимостью от суммы значений в массиве.

Cocktail Sort целесообразно использовать для массивов небольшого размера или почти отсортированных данных, где эффективность алгоритма достигает О(N).

Неар Sort подходит для массивов любого размера, особенно если требуется

сортировка ”на месте” и высокая производительность для всех типов входных данных. Bead Sort рекомендуется использовать исключительно для неотрицательных целых чисел и небольших массивов с ограниченными значениями элементов (тает), поскольку его память и временные затраты резко возрастают при увеличении входных данных.

4. Заключение

В ходе выполнения лабораторной работы мною были реализованы алгоритмы Cocktail Sort, Heap Sort и Bead Sort. Цель работы была достигнута путём анализа и тестирования алгоритмов на массивах с различным количеством элементов и степенью упорядоченности. Полученные экспериментальные результаты подтвердили теоретические оценки сложности каждого из алгоритмов.

В качестве дальнейших исследований можно предложить оптимизацию алгоритмов с точки зрения использования памяти, например, для Bead Sort, который требует значительных ресурсов при больших значениях элементов массива. Также перспективным направлением является изучение адаптивных версий алгоритмов для специфических входных данных, а для Heap Sort — исследование параллельных реализаций, что позволит улучшить производительность при работе с большими массивами.

6. Приложения

Приложение А.

Листинг кода файла cocktail\_sort.cpp

#include <iostream>

#include "cocktail\_sort.h"

// Функция сортировки

void cocktail\_sort(int arr[], int n) {

    int start = 0; // Точка начала сортировки

    int end = n - 1; // Точка конца сотрировки

    bool swap = true; // Переменная для определения, была ли перестановка

    while (swap) {

        swap = false;

        // Делаем сортировку слева на правло

        for (int i = start; i < end; i++){

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

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

                swap = true;

            }

        }

        // Если ничего не поменялось местами, то сортировка прошла успешно

        if (!swap) {

            break;

        }

        swap = false;

        end--; // Сдвигаем правую границу

        // Делаем сортировку справа на лево

        for (int i = end - 1; i >= start; i--) {

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

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

                swap = true;

            }

        }

        // Если ничего не поменялось местами, то сортировка прошла успешно

        if (!swap) {

            break;

        }

        start++; // Сдвигаем левую границу

    }

}

Приложение B.

Листинг кода файла heap\_sort.cpp

#include <iostream>

#include "heap\_sort.h"

// Функция для "просеивания" элемента в куче

void heapify(int arr[], int n, int i) {

    int largest = i;  // Инициализируем корень как наибольший

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

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

    // Если левый элемент больше корня

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

        largest = left;

    }

    // Если правый элемент больше

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

        largest = right;

    }

    // Если наибольший элемент не корень

    if (largest != i) {

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

        // Рекурсивно "просеиваем" затронутый поддерево

        heapify(arr, n, largest);

    }

}

// Основная функция для сортировки

void heapsort(int arr[], int n) {

    // Строим кучу (перестроение массива в кучу)

    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 >= 1; i--) {

        // Перемещаем текущий корень (максимум) в конец массива

        std::swap(arr[0], arr[i]);

        // Вызываем heapify на уменьшенной куче

        heapify(arr, i, 0);

    }

}

Приложение C.

Листинг кода файла bead\_sort.cpp

#include <iostream>

#include <algorithm>

#include <vector>

#include "bead\_sort.h"

std::vector<int> bead\_sort(std::vector<int> a, int len) {

    // Максимальный элемент

    int max = a[0];

    for (int i = 1; i < len; i++) {

        if (a[i] > max) {

            max = a[i];

        }

    }

    // Выделение места для бусин

    std::vector<std::vector<int>> beads;

    beads.resize(len);

    for (int i = 0; i < len; i++) {

        beads[i].resize(max);

        std::fill(beads[i].begin(), beads[i].end(), 0); // массив заполняется нулями

    }

    // Помечаем бусины

    // Мы "визуализируем" бусины для каждого элемента из массива

    for (int i = 0; i < len; i++) {

        for (int j = 0; j < a[i]; j++) {

            beads[i][j] = 1;

        }

    }

    // Перемещаем бусины вниз

    // Мы снова "визуализируем", но уже падения бусин

    for (int j = 0; j < max; j++) {

        int sum = 0; // Счет количества буссин в столбце j

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

            sum += beads[i][j];

            beads[i][j] = 0;

        }

        // "Перемещаем" бусини вниз

        for (int i = len-1; i >= len-sum; i--) {

            beads[i][j] = 1;

        }

    }

    // Получение отсортированного массива

    std::vector<int> result;

    result.resize(len);

    // Счет количества бусин в каждом столбце (суммируем все элементы в строке)

    for (int i = 0; i < len; i++) {

        int sum = 0;

        for (int j = 0; j < max; j++) {

            sum += beads[i][j];

        }

        result[i] = sum;

    }

    // Получаем отсортированное значение

    return result;

}

Приложение D.

Листинг кода файла test\_sort.cpp

#include "cocktail\_sort.h"

#include "heap\_sort.h"

#include "bead\_sort.h"

#include <cassert>

#include <iostream>

#include <vector>

#include <algorithm>

bool is\_sorted(const std::vector<int>& arr) {

    for (size\_t i = 1; i < arr.size(); ++i) {

        if (arr[i - 1] > arr[i]) {

            return false;

        }

    }

    return true;

}

void test\_cocktail\_sort() {

    int best\_case[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    int n = sizeof(best\_case) / sizeof(best\_case[0]);

    cocktail\_sort(best\_case, n);

    assert(is\_sorted(std::vector<int>(best\_case, best\_case + n)));

    int average\_case[] = {5, 3, 8, 6, 2, 7, 4, 1, 9, 10};

    cocktail\_sort(average\_case, n);

    assert(is\_sorted(std::vector<int>(average\_case, average\_case + n)));

    int worst\_case[] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

    cocktail\_sort(worst\_case, n);

    assert(is\_sorted(std::vector<int>(worst\_case, worst\_case + n)));

    std::cout << "Cocktail Sort tests passed!\n";

}

void test\_heap\_sort() {

    int best\_case[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    int n = sizeof(best\_case) / sizeof(best\_case[0]);

    heapsort(best\_case, n);

    assert(is\_sorted(std::vector<int>(best\_case, best\_case + n)));

    int average\_case[] = {5, 3, 8, 6, 2, 7, 4, 1, 9, 10};

    heapsort(average\_case, n);

    assert(is\_sorted(std::vector<int>(average\_case, average\_case + n)));

    int worst\_case[] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

    heapsort(worst\_case, n);

    assert(is\_sorted(std::vector<int>(worst\_case, worst\_case + n)));

    std::cout << "Heap Sort tests passed!\n";

}

void test\_bead\_sort() {

    std::vector<int> best\_case = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    auto result\_best = bead\_sort(best\_case, best\_case.size());

    assert(is\_sorted(result\_best));

    std::vector<int> average\_case = {5, 3, 8, 6, 2, 7, 4, 1, 9, 10};

    auto result\_avg = bead\_sort(average\_case, average\_case.size());

    assert(is\_sorted(result\_avg));

    std::vector<int> worst\_case = {10, 10, 10, 10, 10};

    auto result\_worst = bead\_sort(worst\_case, worst\_case.size());

    assert(is\_sorted(result\_worst));

    std::cout << "Bead Sort tests passed!\n";

}

int main() {

    test\_cocktail\_sort();

    test\_heap\_sort();

    test\_bead\_sort();

    std::cout << "All tests passed successfully!\n";

    return 0;

}