ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 5

«Сортировка»

Выполнил работу

Бабич Александр

Академическая группа №J3111

Принято

Ментор, Вершинин Владислав

Санкт-Петербург

2024

Структура отчёта:

1. Введение

Основная цель — научиться находить и исследовать необходимую информацию.

Задача — найти, реализовать и изучить 3 вида сортировок.

2. Теоретическая подготовка

Необходимо знать как пишутся и из чего состоят: циклическая сортировка, сортировка кучей, сортировка по корзинам.

3. Реализация

В данной работе были реализованы и протестированы три алгоритма сортировки: Cycle Sort, Heap Sort и Bucket Sort.

Cycle Sort: Алгоритм минимизирует количество записей, размещая каждый элемент на его "правильное" место в массиве. Он проходит по массиву и вычисляет, где каждый элемент должен находиться в отсортированном массиве, затем циклически перемещает элементы на нужные позиции до тех пор, пока все элементы не будут упорядочены.

Heap Sort: Этот алгоритм строит двоичную кучу (max-heap), чтобы находить максимальный элемент в массиве. Затем максимальный элемент перемещается в конец массива, и оставшаяся часть массива перестраивается снова как куча. Этот процесс повторяется, пока весь массив не будет отсортирован.

Bucket Sort: Алгоритм распределяет элементы по "корзинам" (buckets) на основе их значений. Каждая корзина затем сортируется отдельно с помощью Insertion Sort. После этого отсортированные корзины объединяются, формируя окончательно отсортированный массив.

4.Экспериментальная часть

Асимптотика Cycle Sort: Временная сложность:

1. В худшем случае (если массив отсортирован в обратном порядке) Cycle Sort требует выполнения нескольких циклов по массиву,

что приводит к временной сложности O(n^2), где n — количество элементов в массиве.

2. В среднем случае алгоритм также выполняет O(n^2) операций, поскольку на каждом цикле поиска позиции элемента мы проходим по оставшейся части массива.

3. В лучшем случае, если элементы массива уже отсортированы, каждый элемент будет на своей позиции, и цикл завершится после первой перестановки, что даст временную сложность O(n^2).

Пространственная сложность:

Cycle Sort не использует дополнительных массивов для хранения данных, все операции происходят в самом массиве. Это приводит к пространственной сложности O(1), то есть алгоритм работает с постоянным количеством дополнительной памяти.

Итого:

- Худшая временная сложность: O(n^2)
- Лучшая временная сложность: O(n^2)
- Средняя временная сложность: O(n^2)
- Пространственная сложность: O(1)

Асимптотика пирамидальной сортировки

Временная сложность:

1. Построение кучи (функция heapify) выполняется за время O(n), так как каждый элемент в куче может быть обработан не более чем за O(log n) времени.

Однако, для всего массива этот процесс оценивается как O(n), поскольку большая часть элементов будет обрабатываться за O(1) или O(log n) по мере продвижения снизу вверх.

2. Сортировка массива путём извлечения элементов из кучи (извлечение максимального элемента и восстановление кучи) требует O(log n) времени для каждого извлечения.

Так как извлекается n элементов, суммарная сложность этого этапа — O(n log n).

Итого, общая временная сложность:

- Худшая временная сложность: O(n log n)
- Средняя временная сложность: O(n log n)
- Лучшая временная сложность: O(n log n)

Пространственная сложность:

- Не требует дополнительной памяти для хранения промежуточных данных, пространственная сложность алгоритма составляет O(1).

Итого:

- Временная сложность: O(n log n)
- Пространственная сложность: O(1)

Асимптотика Bucket Sort

Временная сложность:

1. Разбиение элементов на бакеты — это операция за время O(n), где n — количество элементов в массиве.

Элементы разбиваются по индексам, которые рассчитываются как произведение значения элемента на количество бакетов.

2. Сортировка каждого бакета происходит с использованием сортировки вставками, которая имеет временную сложность O(k^2) для каждого бакета, где k — количество элементов в бакете.

Однако, в худшем случае, все элементы могут попасть в один бакет, что приведет к использованию сортировки вставками для всех элементов, что дает сложность O(n^2).

3. Объединение всех бакетов в один массив также требует O(n) времени.

Итого, общая временная сложность:

- Худшая временная сложность: O(n * k), если все элементы попадают в один бакет, можно n^2(т.к. сортировка вставками может пройтись по всем элементам)

- Средняя временная сложность: O(n * k), где k — количество элементов в бакете.

- Лучшая временная сложность: O(n), Лучший случай происходит, когда каждый блок получает одинаковое количество элементов.

- В этом случае каждый вызов сортировки вставкой будет занимать постоянное время, поскольку количество элементов в каждом блоке будет постоянным (предполагая, что k линейно пропорционально n).

- То есть значением k мы можем пренебречь

Пространственная сложность:

- Пространственная сложность алгоритма сортировки по бакетам составляет O(n * k), где n — количество элементов в массиве, а k — количество бакетов. Пространственная сложность зависит от размера массива и количества бакетов.

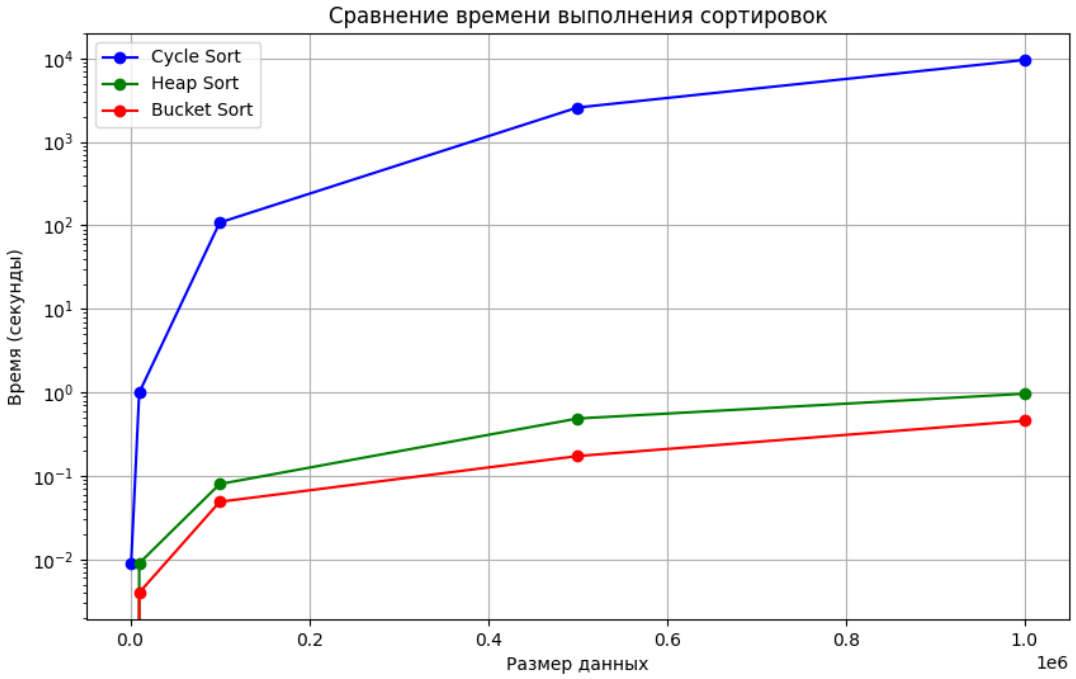
Итого:

- Временная сложность: O(n^2) (в худшем случае), O(n * k) (в лучшем случае)

- Пространственная сложность: O(n * k)

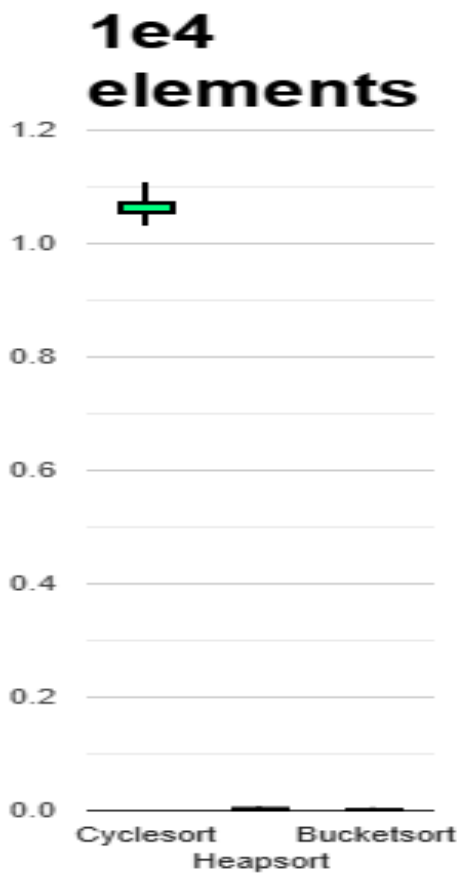
График зависимости времени от числа элементов.

Согласно требованиям моего варианта, на вход к моим алгоритмам подаётся до 1e6 элементов. Для тестирования алгоритма была собрана статистика, приведенная в изображении №1.

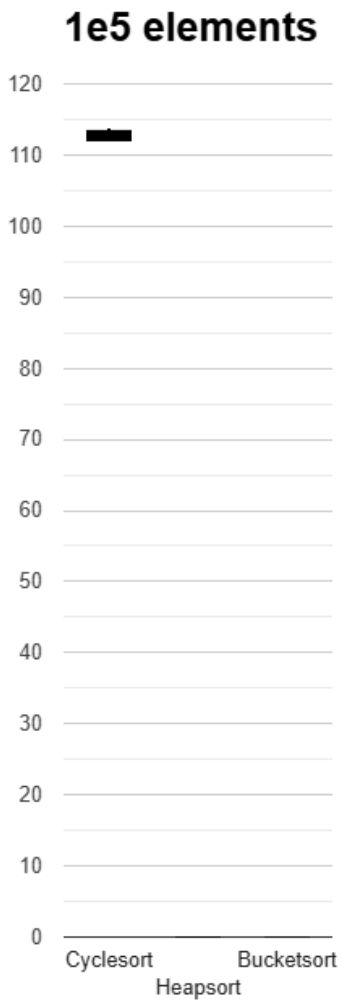


Изображение №1 - График работы алгоритмов

Cyclesort имеет асимптотику o(n^2) и растет сильнее всего, он не эффективен на больших данных.



Изображение №2 - Boxplot график работы алгоритмов на 1e4 элементах



Изображение №3 - Boxplot график работы алгоритмов на 1e5 элементах

5.Заключение

В данной работе были реализованы и проанализированы три алгоритма сортировки: цикличная сортировка, сортировка кучей и сортировка по корзинам. Все три алгоритма имеют разные подходы и асимптотические характеристики, что позволяет использовать их в различных практических случаях в зависимости от структуры и объема данных.

Цикличная сортировка (Cycle Sort) – это алгоритм, разработанный для сортировки с минимальным количеством перемещений. Сложность этого

алгоритма составляет O(n^2), что делает его неэффективным для больших массивов. Однако, его уникальное преимущество заключается в минимальном количестве операций записи, поэтому его можно применять в задачах, где запись данных является затратной операцией или где важна устойчивость к изменениям памяти, например, в ограниченных ресурсах.

Сортировка кучей (Heap Sort) обладает асимптотикой O(n*logn)

и хорошо работает на больших массивах данных. На практике алгоритм демонстрирует стабильные результаты, и выбросы по времени выполнения минимальны, так как структура кучи обеспечивает равномерный порядок работы. Этот алгоритм можно применять для задач, где важны надежность и предсказуемая производительность при работе с массивами среднего и большого размера.

Сортировка по корзинам (Bucket Sort) оптимально работает с данными, равномерно распределенными по определенному диапазону, и имеет асимптотику O(n*k), где k — количество корзин. На практике эффективность алгоритма зависит от правильного выбора количества и ширины корзин: при неравномерном распределении элементов по корзинам могут возникнуть выбросы по времени из-за переполнения отдельных корзин. Этот алгоритм подходит для задач, где известно распределение данных и требуется высокая скорость сортировки на относительно малых объемах данных.

6.Приложения

ПРИЛОЖЕНИЕ А

Листинг кода файла cyclesort.cpp

```
void Swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

void cycleSort(std::vector<int> &array) {
    int n = array.size(); // Получаем размер вектора
    int writes = 0;

    // Проходим по всем элементам и ставим их на правильное место
    for (int cycle_start = 0; cycle_start <= n - 2; cycle_start++) {
        // Инициализируем элемент как начальную точку
        int item = array[cycle_start];

        // Находим позицию, на которую нужно поставить элемент.
        int pos = cycle_start;
        for (int i = cycle_start + 1; i < n; i++)
            if (array[i] < item)
                pos++;

        // Если элемент уже на правильной позиции
        if (pos == cycle_start)
            continue;

        // Игнорируем все одинаковые элементы
        while (item == array[pos])
            pos++;

        // Ставим элемент на его правильную позицию
        if (pos != cycle_start) {
            Swap(item, array[pos]);
            writes++;
        }

        // Поворачиваем остаток цикла
        while (pos != cycle_start) {
            pos = cycle_start;

            // Находим позицию, на которую нужно поставить элемент
            for (int i = cycle_start + 1; i < n; i++)
                if (array[i] < item)
                    pos++;

            // Игнорируем все одинаковые элементы
            while (item == array[pos])
                pos++;

            // Ставим элемент на его правильную позицию
            if (item != array[pos]) {
                Swap(item, array[pos]);
                writes++;
            }
        }
    }
}
```

Приложение Б

Листинг кода файла heapsort.cpp

```cpp
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

// Функция для восстановления свойства кучи
void heapify(std::vector<int> &array, int n, int i) {
    int root = i; // корень
    int left_child = 2 * i + 1; // левый потомок
    int right_child = 2 * i + 2; // правый потомок

    // Проверка левого дочернего элемента
    if (left_child < n && array[left_child] > array[root]) {
        root = left_child;
    }

    // Проверка правого дочернего элемента
    if (right_child < n && array[right_child] > array[root]) {
        root = right_child;
    }

    // Если наибольший элемент не является корнем
    if (root != i) {
        swap(array[i], array[root]);
        heapify(array, n, root);
    }
}

// Функция для сортировки массива методом пирамидальной (кучей) сортировки
void heapSort(std::vector<int> &array) {
    int n = array.size();

    // Построение кучи (перегруппировка массива)
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(array, n, i);
    }

    // Извлечение элементов из кучи по одному
    for (int i = n - 1; i > 0; i--) {
        swap(array[0], array[i]); // Перемещаем текущий корень в конец
        heapify(array, i, 0); // Вызываем heapify для уменьшенной кучи
    }
}
```

Приложение В

Листинг кода файла bucketsort.cpp

```cpp
void bucketSort(std::vector<float> &array) {
    // Находим минимальное и максимальное значения в массиве
    // Это нужно для того, чтобы правильно распределить элементы по бакетам.
    float min = findMin(array);
    float max = findMax(array);

    // Количество бакетов (корзин) зависит от количества элементов в массиве
    // Например, если у нас 10 элементов, создадим 10 бакетов.
    // Можно изменить на любое другое количество бакетов, например, n / 2 или n
/ 10
    int bucketCount = array.size();  // В данном случае количество бакетов =
количеству элементов в массиве

    // Создаем бакеты (корзины) — это массив из векторов
    // Каждый бакет будет хранить значения, которые попадают в его диапазон.
    std::vector<float> buckets[bucketCount];

    // Разделяем элементы массива на бакеты (корзины)
    for (size_t i = 0; i < array.size(); ++i) {
        // Рассчитываем индекс для текущего элемента, который будет определять,
в какой бакет его поместить.
        // Индекс зависит от разницы между значением элемента и минимальным
значением,
        // а также от диапазона возможных значений (от min до max).
        int index = (array[i] - min) * (bucketCount - 1) / (max - min);

        // Помещаем элемент в соответствующий бакет
        // Здесь индекс бакета зависит от значения элемента.
        buckets[index].push_back(array[i]);
    }

    // Сортируем каждый бакет с помощью сортировки вставками
    // В зависимости от того, сколько элементов в каждом бакете, используется
сортировка вставками.
    // Это происходит, потому что в большинстве случаев количество элементов в
бакете
    // будет небольшим (что делает сортировку вставками эффективной).
    for (int i = 0; i < bucketCount; ++i) {
        // Если бакет не пустой, сортируем его
        if (!buckets[i].empty()) {
            insertionSort(buckets[i]);  // Сортировка вставками для каждого
бакета
        }
    }

    // Собираем отсортированные элементы обратно в массив
    // После того как все бакеты отсортированы, объединяем их в один
отсортированный массив.
    size_t index = 0;
    for (int i = 0; i < bucketCount; ++i) {
        for (size_t j = 0; j < buckets[i].size(); ++j) {
            // Сначала заполняем исходный массив отсортированными элементами
            array[index++] = buckets[i][j];
        }
    }
}
```