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

Отчёт по лабораторной работе № 5

«Алгоритмы сортировки»

Выполнил работу

Смирнов Александр

Академическая группа №J3111

Принято

Ментор Вершинин Владислав Константинович

Санкт-Петербург

2024

1. Введение

Цель работы: изучить алгоритмы сортировки и научиться применять их на практике.

Задачи:

1. Выбрать алгоритмы для реализации;
2. Ознакомиться с их устройством;
3. Изучить особенности реализации этих алгоритмов на C++;
4. Реализовать алгоритмы;
5. Провести анализ полученных результатов;
6. Подготовить отчёт.

2. Теоретическая подготовка

В работе я использовал типы данных bool, int, double, vector.

Вспомогательные алгоритмы старался максимально не использовать, чтобы глубже понять суть работы выбранных сортировок и попрактиковаться в написании кода на C++.

3. Реализация

Я начал работу с поиска подходящих по сложности алгоритмов сортировок. Я выбрал shaker sort, timsort и radix sort, так как они показались мне интересными и не были разобраны на лекции.

Реализовал функции этих сортировок и вспомогательные функции для них, функции тестирования на корректность и скорость алгоритмов, и main функцию для запуска всех тестов и вывода результатов.

Я использовал библиотеки:

iostream, vector, time, cassert, algorithm и cmath.

3.1 Shaker Sort – O(n^2)

Shaker Sort – модификация алгоритма сортировки пузырьком. Он проходит по массиву сначала слева направо, затем справа налево. При первом проходе наибольший элемент оказывается в конце массива, а наименьший – в начале, что делает алгоритм более эффективным, чем простая сортировка пузырьком. Так выглядит код данной сортировки (важным элементом является проверка на то, отсортирован ли массив, ведь в крайних случаях она может значительно ускорить процесс выполнения алгоритма, например, для уже отсортированных массивов) (рис. 1):

```
void shakerSort(std::vector<int>& arr) {
    int left = 0; // Начало массива, 4 байта
    int right = arr.size() - 1; // Конец массива, 4 байта
    bool swapped; // Флаг, указывающий на то, было ли произведено обмен, 1 байт

    do { // O(n) в худшем случае
        swapped = false;

        // Проходим по массиву слева направо
        for (int i = left; i < right; ++i) { // O(n) в худшем случае
            if (arr[i] > arr[i + 1]) {
                std::swap(arr[i], arr[i + 1]); // Обмен элементов
                swapped = true; // Установим флаг, если произошел обмен
            }
        }
        // Уменьшаем правую границу, так как последний элемент уже на месте
        right--;

        // Если не было обменов, массив уже отсортирован
        if (!swapped) {
            break;
        }

        swapped = false;

        // Проходим по массиву справа налево
        for (int i = right; i > left; --i) { // O(n) в худшем случае
            if (arr[i] < arr[i - 1]) {
                std::swap(arr[i], arr[i - 1]); // Обмен элементов
                swapped = true; // Установим флаг, если произошел обмен
            }
        }
        // Увеличиваем левую границу, так как первый элемент уже на месте
        left++;

    } while (left <= right); // Повторяем, пока левая граница не превысит правую
}
```

Рисунок 1 Код функции shakerSort с подробными комментариями

Оценочная сложность для худшего случая – O(n). Важным плюсом является то, что функция почти не потребляет память благодаря использованию ссылки. Их я использую в дальнейшем для каждой функции сортировки.

3.2. Timsort – O(n * log(n))

Timsort — гибридный алгоритм сортировки, который сочетает сортировку вставками и сортировку слиянием.

Работа алгоритма состоит из трёх фаз:

1. Вычисление длины подмассива. Входной массив делят на подмассивы, для этого задается параметр RUN, задающий минимальный размер подмассива. В интернете рекомендуют задавать этот параметр в диапазоне от 32 до 64. Слишком маленький RUN: Это может привести к более частой сортировке вставками, что увеличивает общее время сортировки. Слишком большой RUN: в этом случае меньше подмассивов, и эффективность слияния может быть снижена.
2. Сортировка каждого подмассива вставками. К каждому подмассиву применяют сортировку вставками. Так как размер подмассива невелик и часть его уже упорядочена, сортировка работает быстро и эффективно.
3. Слияние отсортированных подмассивов в единый массив с помощью модифицированной сортировки слиянием.

На первом этапе находится наименьший элемент в неотсортированной части и помещается в начало отсортированной части. Затем ищем следующий наименьший элемент и помещаем его сразу после первого. Этот процесс продолжается до тех пор, пока все элементы из неотсортированной части не будут помещены в отсортированную.

На втором этапе берем две или более отсортированные части и сливаем их в одну большую отсортированную часть. Этот процесс повторяется до тех пор, пока не будет отсортирован весь массив.

Далее представлены функции для реализации этих этапов (рис. 2, 3):

```cpp
// Функция сортировки вставками для сортировки небольших массивов
void insertionSort(std::vector<int>& arr, int left, int right) {
    for (int i = left + 1; i <= right; i++) { // O(right - left + 1)
        int key = arr[i]; // 4 байта
        int j = i - 1; // 4 байта

        // Вставляем arr[i] в отсортированную часть
        while (j >= left && arr[j] > key) { // O(right - left + 1)
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
```

Рисунок 2. Код функции сортировки подмассива вставками

```cpp
// Функция слияния двух отсортированных "бегов"
void merge(std::vector<int>& arr, int left, int mid, int right) {
    // Находим размеры двух подсортированных массивов
    int len1 = mid - left + 1;
    int len2 = right - mid;
    // Все по 4 байта

    // Создаем временные массивы
    std::vector<int> leftArray(len1);
    std::vector<int> rightArray(len2);
    // Все по 24 + 4n байт

    // Заполняем временные массивы
    for (int i = 0; i < len1; i++) // O(len1)
        leftArray[i] = arr[left + i];
    for (int j = 0; j < len2; j++) // O(len2)
        rightArray[j] = arr[mid + 1 + j];

    // Слияние временных массивов обратно в основной массив
    int i = 0; // Начальный индекс первого подмассива
    int j = 0; // Начальный индекс второго подмассива
    int k = left; // Начальный индекс для слияния
    // Все по 4 байта

    while (i < len1 && j < len2) { // O(len1 + len2)
        if (leftArray[i] <= rightArray[j]) {
            arr[k] = leftArray[i];
            i++;
        } else {
            arr[k] = rightArray[j];
            j++;
        }
        k++;
    }

    // Копируем оставшиеся элементы, если есть
    while (i < len1) { // O(len1)
        arr[k] = leftArray[i];
        i++;
        k++;
    }
    while (j < len2) { // O(len2)
        arr[k] = rightArray[j];
        j++;
        k++;
    }
}
```

Рисунок 3. Код функции слияния отсортированных подмассивов (рис. 4)

```
// Основная функция Timsort
void timSort(std::vector<int>& arr) {
    int n = arr.size(); // 4 байта

    // Сортировка подмассивов (бегов) длины RUN
    for (int start = 0; start < n; start += RUN) { // O(n)
        int end = std::min(start + RUN - 1, n - 1); // 4 байта
        insertionSort(arr, start, end);
    }

    // Слияние подмассивов в порядке возрастания
    for (int size = RUN; size < n; size *= 2) { // O(log(n))
        for (int left = 0; left < n; left += 2 * size) { // O(n)
            int mid = std::min(left + size - 1, n - 1);
            int right = std::min((left + 2 * size - 1), (n - 1));

            // Слияние подмассивов
            if (mid < right) {
                merge(arr, left, mid, right);
            }
        }
    }
}
```

Рисунок 4. Код основной функции Timsort

3.3. Radix sort – O(n * max/10)

Сортировка Radix Sort работает следующим образом:

1. Начинается с наименее значащей цифры (правой).
2. Сортирует значения на основе этой цифры, помещая элементы в правильную корзину в зависимости от неё, а затем возвращая их в массив в правильном порядке.
3. Переходит к следующей цифре и снова сортирует, как на предыдущем этапе, пока не останутся только значащие цифры.

Алгоритм работает так: сначала сравниваются значения одного крайнего разряда, и элементы группируются по результатам этого сравнения. Затем сравниваются значения следующего разряда, соседнего, и элементы либо упорядочиваются по результатам сравнения значений этого разряда внутри образованных на предыдущем проходе групп, либо переупорядочиваются в целом, но сохраняя относительный порядок, достигнутый при предыдущей

сортировке. Затем аналогично делается для следующего разряда, и так до конца.

Так как выравнивать сравниваемые записи относительно друг друга можно в разную сторону, на практике существуют два варианта этой сортировки: можно выровнять записи чисел в сторону менее значащих цифр (по правой стороне, в сторону единиц — LSD от англ. least significant digit) или более значащих цифр (по левой стороне, со стороны более значащих разрядов — MSD от most significant digit).

Ниже представлен код алгоритма этой сортировки (рис. 5, 6):

```
// Функция для получения i-й цифры числа
int getDigit(int number, int digitPosition) {
    return (number / (int)pow(10, digitPosition)) % 10;
}


// Функция сортировки, использующая Counting Sort на основании разряда
void countingSort(std::vector<int>& arr, int digitPosition) {
    const int base = 10; // Мы используем десятичную систему, 4 байта
    int n = arr.size(); // 4 байта

    std::vector<int> output(n); // Временный вектор для отсортированных значений, 24 + 4n байт
    std::vector<int> count(base, 0); // Вектор счетчиков для каждой цифры, 24 + 4 * 10 = 64 байта

    // Подсчитываем количество элементов по текущему разряду
    for (int i = 0; i < n; i++) { // O(n)
        int digit = getDigit(arr[i], digitPosition);
        count[digit]++;
    }

    // Изменяем count[i] так, чтобы count[i] содержал позицию этого разряда в выходном массиве
    for (int i = 1; i < base; i++) { // всего 9 итераций
        count[i] += count[i - 1];
    }

    // Строим выходной массив
    for (int i = n - 1; i >= 0; i--) { // O(n)
        int digit = getDigit(arr[i], digitPosition); // 4 байта
        output[count[digit] - 1] = arr[i];
        count[digit]--;
    }

    // Копируем отсортированные элементы обратно в исходный массив
    for (int i = 0; i < n; i++) { // O(n)
        arr[i] = output[i];
    }
}
```

Рисунок 5. Вспомогательные функции для сортировки radix sort с комментариями

```
// Основная функция Radix Sort
void radixSort(std::vector<int>& arr) {
    // Найдем максимальное число, чтобы знать, сколько разрядов нам нужно сортировать
    int maxElement = *max_element(arr.begin(), arr.end()); // O(n)

    // Определим количество разрядов
    int digitPosition = 0;
    while (maxElement > 0) { // O(maxElement / 10)
        countingSort(arr, digitPosition);
        maxElement /= 10;
        digitPosition++;
    }
}
```

Рисунок 6. Основная функция radix sort

Я измерял скорость работы этих алгоритмах на массивах одинаковой длинны от 1000 до 10000 элементов с шагом 1000 со значениями от 0 до 99. Результаты измерений на картинках ниже (рис. 7, таблица 1)

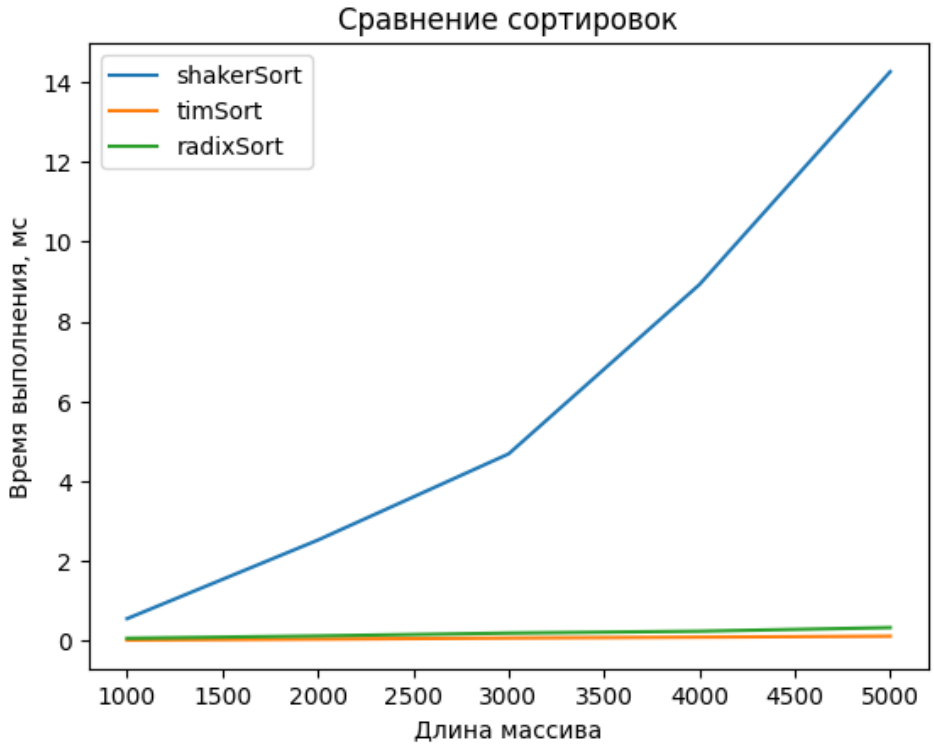

Рисунок 7. График изменения скорости разных сортировок в зависимости от длинны входного массива

N	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
shakerSort	0.5588	2.52770	4.68730	8.92730	14.26110	19.44750	26.34120	35.16810	43.74290	53.58440
timSort	0.0215	0.0461	0.0734	0.0956	0.1204	0.1452	0.1701	0.2369	0.2432	0.2730
radixSort	0.0682	0.1268	0.198	0.245	0.3346	0.3862	0.4214	0.4804	0.543	0.5989

Таблица 1. Результаты измерений в виде таблицы (время в мс)

Также изучил распределения времени выполнения различных сортировок на 50 разных массивах длины 1000 (рис. 8):

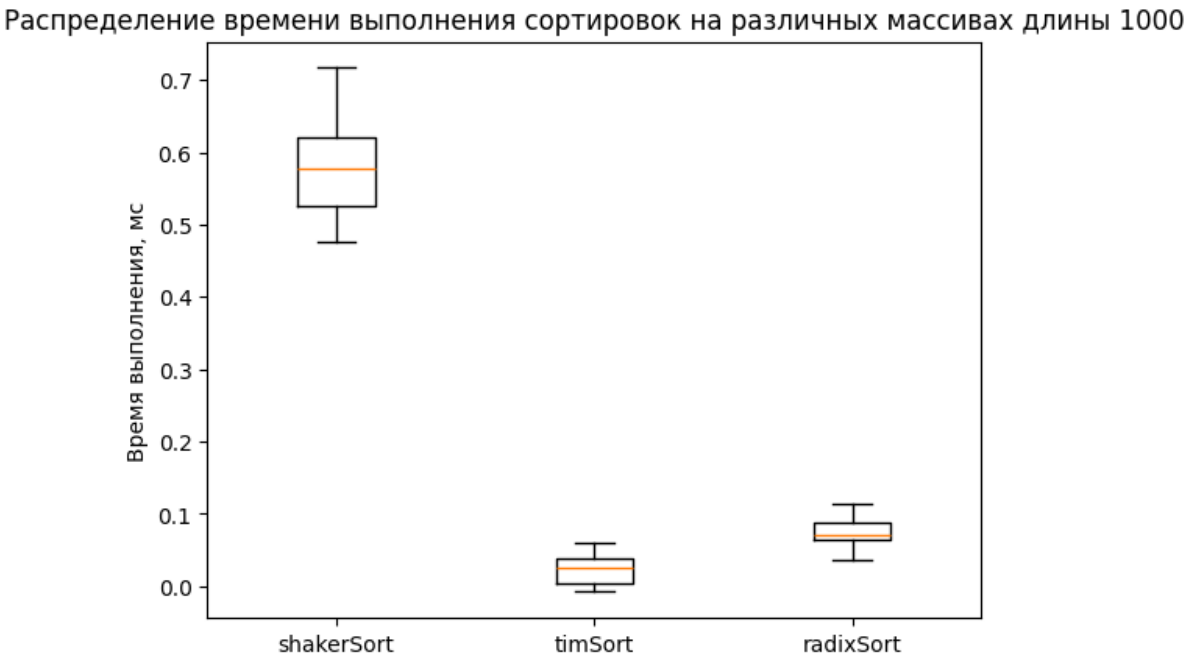

Рисунок 8. Распределения времени выполнения.

Вы итоге сортировка Timsort оказалась самой эффективной и стабильной сортировкой по времени выполнения из трех выбранных мной.

5. Заключение

В ходе выполнения работы мною о реализовано 3 алгоритма сортировок массива. Цель работы была достигнута путём тестирования на массивах с различным количеством элементов. Полученные результаты также совпадают с теоретическими оценками сложности алгоритмов, рассчитанными в коде.

В качестве дальнейших исследований можно предложить оптимизацию предложенных алгоритмов по памяти, возможно попробовать комбинировать различные подходы к сортировкам (как в Timsort).

6. Приложения

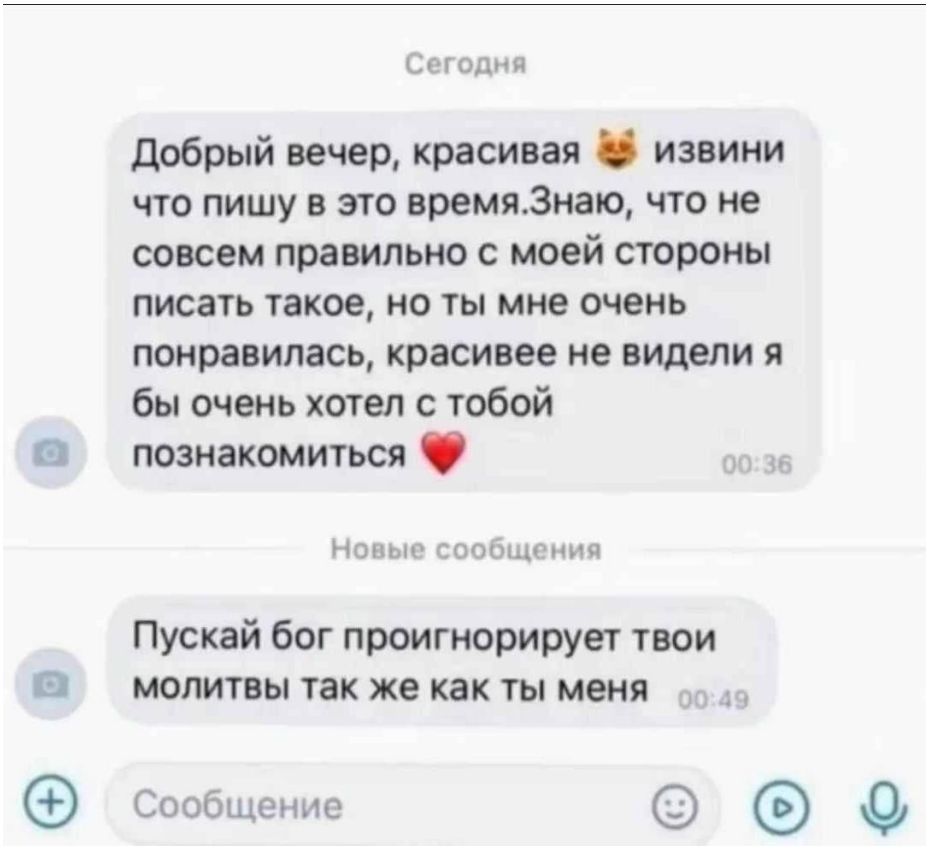

Рисунок 9. Это я так расстраиваюсь, когда месяц жду проверку 4 лабы