Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное   
учреждение высшего образования

Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

**Отчет по лабораторной работе**

**«Сортировки»**

**Выполнил**:

студент группы 3822Б1ПМ1

Борисов С. М.

**Проверил**:

преподаватель каф. МОСТ,

Волокитин В.Д.

Нижний Новгород

2022

**Содержание**

[Постановка задачи 3](#_Toc26962562)

[Метод решения 4](#_Toc26962563)

[Руководство пользователя 5](#_Toc26962564)

[Описание программной реализации 6](#_Toc26962565)

[Подтверждение корректности 7](#_Toc26962566)

[Результаты экспериментов 8](#_Toc26962567)

[Заключение 9](#_Toc26962568)

[Приложение 10](#_Toc26962569)

# Постановка задачи

Цель – изучение различных алгоритмов сортировок (работающих за O(n²), O(n) и O(n))

Задачи, необходимые для достижения цели:

1. Изучение алгоритмов упорядочивания массивов с числами с плавающей запятой
2. Написание алгоритмов сортировки
3. Написание пользовательского интерфейса
4. Сравнить скорость и эффективность этих сортировок, корректность работы
5. Проверить соответствие теоретической оценке выполнения

# Метод решения

Выбраны следующие виды сортировок:

1. Сортировка пузырьком
2. Сортировка Хоара
3. Поразрядная сортировка

Далее подробное описание каждой из сортировок:

Сортировка пузырьком:

Основная идея заключается в том, что сначала мы рассматриваем первый элемент массива, сравнивая его с соседним, в зависимости от реализации (по возрастанию или по убыванию) меняем или не меняем местами так, чтобы они встали в необходимом нам порядке. Затем проделываем аналогичную операцию со вторым элементом (он может оказаться и первым элементом изначального массив, если происходила перестановка), затем проделываем всё это с остальными элементами. После первого прохода на нужное место встанет либо наименьший, либо наибольший элемент (зависит от реализации). Из-за такой идеи сортировка и получила название - элементы “всплывают” к началу (концу) массива. На втором проходе всё делаем аналогично, за исключением того, что не требуется сравнивать с последним элементом, так как уже известно, что там стоит необходимый нам. В итоге можно утверждать, что массив отсортирован, потому что предпоследнему элементу необходимо только одно сравнение (все выше уже отсортированы), а последний не нужно сравнивать ни с кем вовсе, потому что он уже стоит на своём месте. Таким образом каждый раз мы делаем количество сравнений, равное количеству элементов в массиве (n) – рассматриваем нами элемент (i), а делаем мы это для всех элементов, то есть выполняется примерно (n-1)\*(n-1), откуда мы и получаем асимптотику О(n²). Данная сортировка имеет дополнение в виде возможности добавления флага, который определяет, совершались ли перестановки. Действительно, если все элементы стоят на своём месте, то при сравнении мы не совершим ни одной перестановки, значит проверив флаг мы можем остановить программу с успешно отсортированным массивом.

Сортировка Хоара (Quick Sort):

Основная идея заключается в том, что мы выбираем ведущий элемент, слева от которого элементы меньше него, а справа больше или равные (может быть наоборот, если необходима сортировка по убыванию). Затем в каждой половине аналогично выбирается ведущий элемент, а справа и слева от него оказываются элементы меньше и больше него соответственно. Одной из самых простых реализаций является рекурсивная, потому что каждый раз мы выполняем упомянутое выше действие для разных подмассивов. Таким образом каждый раз ведущий элемент оказывается на своём месте (кол-во элементов до него и после соответствует необходимому в отсортированном), а так как количество элементов в массивах конечно – мы гарантированно придём к верному результату (отсортированному массиву). Очевидно, что из-за своей реализации данная сортировка будет плохо работать на уже отсортированных массивах (независимо от порядка элементов всё равно будет выбирается элемент). С идеей работы и связана его сложность, так как мы рассматриваем разделения на 2 части, а рассматриваем мы все элементы – выходит O(n).

Линейная сортировка (LSD):

Основная идея заключается в том, что мы не сравниваем числа, а подсчётом выставляем числа в нужном нам порядке. Особенностью является то, что нам необходимо знать тип данных, который мы сортируем. Рассмотрим целые числа. В их записи задействуется m разрядов (конечное количество). Подсчитаем количество каждой из цифр в первом разряде числа (всего 10 цифр: 0-9), а затем расположем числа в нужном нам порядке по последнему слагаемому (по возрастанию или убыванию), при этом если цифры равны, то числа местами не меняем. Проведём такую же операцию с цифрами во втором разряде каждого числа, расположив все элементы массива в нужном порядке по этой цифре, проведя такую операцию m раз, где m – количество слагаемых, мы получим отсортированный массив. А работает это поскольку переходя к новому разряду мы не теряем упорядоченность, которой достигли ранее. Таким образом в худшем случае будет выполнено m\*n операций, где n – кол-во элементов в массиве, отсюда и асимптотика О(n). Понимая как устроены другие типы данных в памяти можно сортировать и их, например тип double кодируется 8 -ью байтами и записывается в обратном порядке. В прямом порядке на 1 бит – знак (1, если “-“; 0 если “+”), 11 бит – порядок, 52 бит – мантисса (само число). Отсюда видно, что работать с числами можно как с int. Нюансом является первый 1 бит, в котором учитывается знак, то есть все отрицательные числа больше положительных. При сортировке сначала будут расположены положительные числа, затем отрицательные, по убыванию (так как используется обратный код), поэтому необходимо учесть это и запить отрицательные числа в правильном порядке до положительных. Поскольку числа в памяти хранятся в обратном порядке необходимо начать с 1 байта, считав его как char (приведя к небольшому типу данных), а далее работать к с int

# Руководство пользователя

При запуске программы перед пользователем появляется меню с возможностью выбора действия. Для выбора необходимо ввести номер необходимой команды

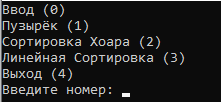


Рисунок 1 – Меню

В данном меню находится 5 команд:

1. Ввод размера массива (генерируется массив псевдослучайных чисел этой длины)
2. Сортировка пузырьком
3. Сортировка Хоара
4. Линейная сортировка
5. Выход – завершает программу

При выборе любой сортировки вернётся меню, показывающее часть исходного массива, часть итогового, длину, а также время выполнения программы

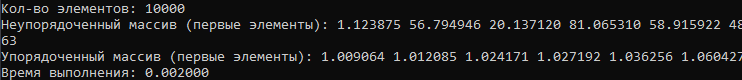


Рисунок 2 - Вывод результата

При вводе номера команды, которой не существует будет выведена соответствующее предупреждение, а после выполнения любой команды вернётся исходное меню, а массив вернётся к неотсортированному виду

# Описание программной реализации

Программа состоит из единственного файла (LabSorted.cpp)

В которой и описаны все функции (14 шт)

void swaps(double\* a, double\* b) – на вход принимает 2 указателя double, меняет местами значения

void printArray(double b[], double nb) – принимает на вход указатель на начало массива и его длину – печатает массив

int compare(const void\* a, const void\* b) – принимает указатели на 2 числа, необходима для работы qsort из stdlib (проверки собственной сортировки) – возвращает необходимые для сортировки значения

int check(double a[], double b[], int len) – принимает указатели на начало отсортированного и изначального массивов, а также длину, проверяет корректно ли сработала моя сортировка – возвращает 1, если да, 0, если нет

void bubble(double\* mas, int n) – принимает указатель на начало массива и его длину, сортирует пузырьком

int partition(double\* mas, int l, int r) – принимает указатель на начало массива, левую и правую границы отрезков. Необходима для работы сортировки Хоара – по относительно выбранного элемента переносит элементы больше и меньше в нужные стороны – возвращает куда сместился выбранный элемент

void quick\_sort(double\* mas, int l, int r) – принимает указатель на начало массива, левую и правую границы отрезков, необходима для рекурсии в сортировке Хоара.

void countByte(unsigned long long\* mas, int size, int count[256], int byte) – принимает указатель на начало массива, его размер, массив определённого размера, рассматриваемый байт – проводит подсчёт цифр в определённом разряде (необходимо для линейной сортировки)

void radix\_sort(unsigned long long\* mas, int size, unsigned long long\* mas\_tmp) – принимает указатели на начала двух массивов, а также размер исходного – выполняет линейную сортировку целых типов данных

void radixDouble(double\* mas, int n, double\* mas\_tmp) - принимает указатели на начала двух массивов, а также размер исходного – выполняет линейную сортировку для double

void input(double\* a, double\* b, int len) - принимает указатели на начала двух массивов, а также размер – заполняет псевдослучайными числами

int output(double a[], double b[], int len, double time) – принимает указатели на начала двух массивов (отсортированный и изначальный), размер, а также время выполнения программы – выводит результат

int MenuChoice(char\* menu[], int ns) – принимает указатель на начало массива с элементами меню, а также их кол-во – служит для вывода меню и выбора элемента

void main() – основная функция, отвечает за корректную работу, использование подпрограмм, взаимодействие с пользователем

# Подтверждение корректности

Для подтверждения корректности в программе была реализована функция check, которая по исходному массиву генерирует такой же, а затем сортирует его встроенной в stdlib.h функцией qsort, которая принимает массив, тип его данных и компоратор (критерий, по которому сортируется массив), а затем поэлементно сравнивает отсортированный мной и этой функцией массив. Если всё корректно – программа выведет результат моей сортировки, иначе выведет ошибку, что отсортировать массив не удалось.

# Результаты экспериментов

По данным экспериментов видно, что сортировка пузырьком показала самый плохой результат. Быстрей работает сортировка Хоара, начиная с 5 миллиона элементов она начала подниматься, а лучшее время показа поразрядная сортировка, которая начала увеличивать время выполнения лишь при 10 миллионах элементов.

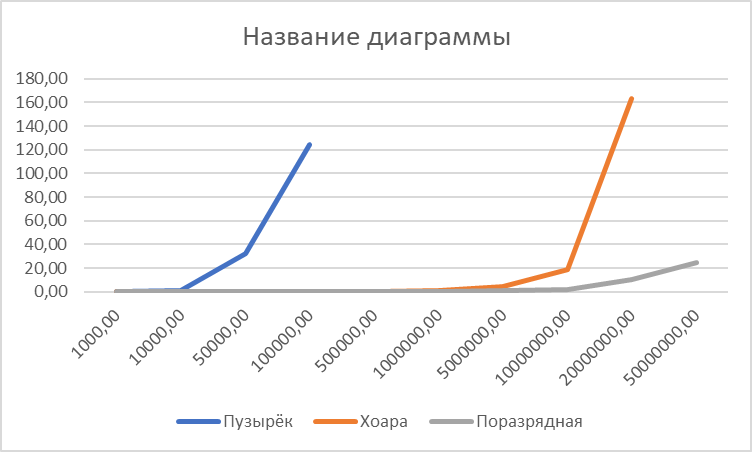


Рисунок 3 - Зависимость времени от кол-ва элементов

Также можно доказать, что теоретическая сложность действительно является такой, для этого необходимо поделить время выполнения программы на сложность и в случае, если всё верно, получим константу, которую мы отбрасываем, для этого будут построены диаграммы (внизу сложность, слева – время, делённое на сложность)

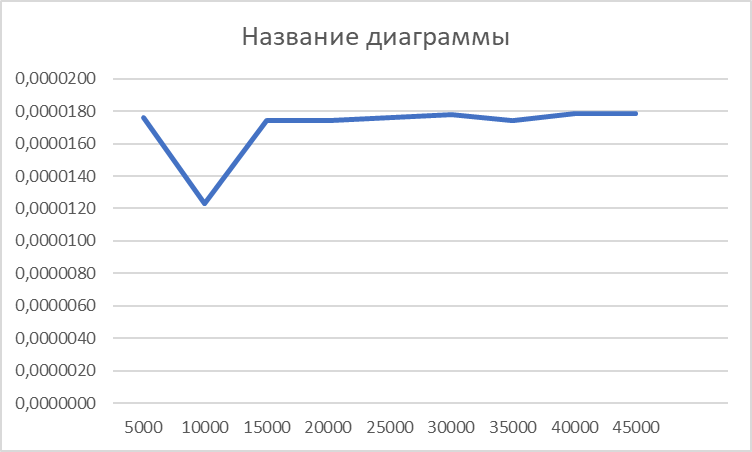


Рисунок 4 – Пузырёк

Видно, что график вышел на константу – значит функция соответствует прогнозируемой сложности

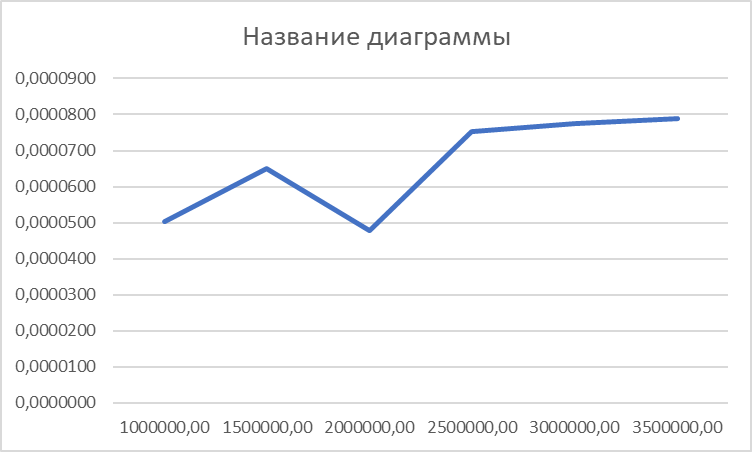


Рисунок 4 – Сортировка Хоара

Возьмём основание логарифма за 2, в таком случае график сходится к константе, значит сложность так же определена верно

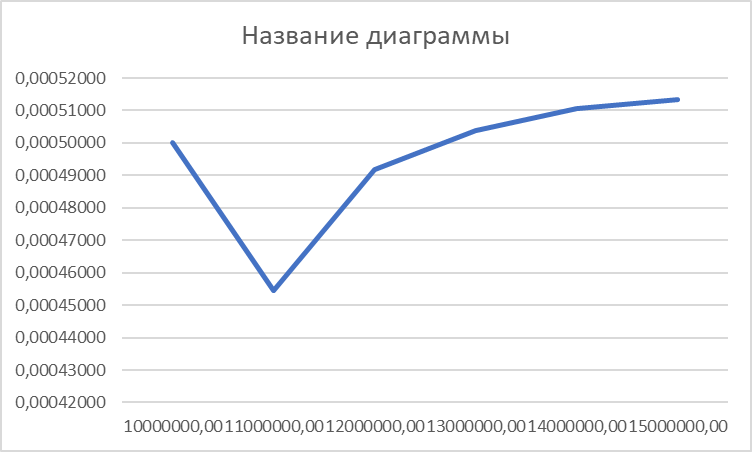


Рисунок 4 – Поразрядная сортировка

В данном случае видно, что график стремится к константе, а значит сложно верна

# Заключение

В результате работы была реализована сортировка для чисел, с плавающей точкой, подробно разобраны такие виды, как сортировка Пузырьком, сортировка Хоара и поразрядная сортировка, для работы с которыми был реализован пользовательский интерфейс. Также была сравнена скорость и эффективность работы этих сортировок. Сортировка Пузырьком – наименее эффективна на больших массивах, далее идёт сортировка Хоара, которая имеет среднюю эффективность из этих 3-х сортировок, самой лучшей же оказалась поразрядная сортировка.

Также было проверено соответствие теоретической асимптотики на практике. Для всех сортировок была получена константа при делении времени выполнения, на сложность алгоритма. Для пузырька C ≈ 0,000018, для Хоара C ≈ 0,00008, а для поразрядной сортировки C ≈ 0,0005

# Приложение

// Hoare

long long int partition(double\* mas, long long int l, long long int r) {

double mid = mas[l + (r - l) / 2];

while (l < r) {

while (mas[l] < mid && (l != r))

l++;

while (mas[r] > mid && (r != l))

r--;

if (l < r) {

if (mas[l] > mas[r]) {

swaps(&mas[l], &mas[r]);

}

else if (mas[l] == mas[r]) { // без этого возможно зацикливание

r--;

}

}

}

return r;

}

void quick\_sort(double\* mas, long long int l, long long int r) {

if (l <= r) {

long long int p = partition(mas, l, r);

quick\_sort(mas, l, p - 1);

quick\_sort(mas, p + 1, r);

}

}

// Bubble

void bubble(double\* mas, long long int n) {

int t = 0;

for (long long int i = 0; i < n - 1; i++) {

for (long long int j = 0; j < n - 1 - i; j++) {

if (mas[j] > mas[j + 1]) {

swaps(&mas[j], &mas[j + 1]);

t = 1;

}

}

if (!t) {

break;

}

}

}

// Radix sort

void countByte(unsigned long long\* mas, long long int size, int count[256], int byte) {

unsigned char\* masc;

int bias;

int tmp1, tmp2;

bias = sizeof(unsigned long long);

masc = (unsigned char\*)mas;

for (int i = 0; i < 256; i++) {

count[i] = 0;

}

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

count[masc[i \* bias + byte]]++;

}

tmp1 = count[0];

count[0] = 0;

for (long long int i = 1; i < 256; i++) {

tmp2 = count[i];

count[i] = count[i - 1] + tmp1;

tmp1 = tmp2;

}

}

void radix\_sort(unsigned long long\* mas, long long int size, unsigned long long\* mas\_tmp) {

unsigned char\* masc = (unsigned char\*)mas;

int count[256];

int sizetype = sizeof(unsigned long long);

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

countByte(mas, size, count, i);

for (long long int j = 0; j < size; j++) {

mas\_tmp[count[masc[j \* sizetype + i]]++] = mas[j];

}

for (long long int j = 0; j < size; j++) {

mas[j] = mas\_tmp[j];

}

}

}

void radixDouble(double\* mas, long long int n, double\* mas\_tmp) {

int count = 0;

long long int i = 0;

radix\_sort((unsigned long long int\*) mas, n, (unsigned long long int\*) mas\_tmp);

while (i < n) {

if (mas[i] >= 0) {

count++;

}

i++;

}

//printf("%d ", count);

for (i = 0; i < count; i++)

mas\_tmp[i + (n - count)] = mas[i];

for (; i < n; i++)

mas\_tmp[n - (++count)] = mas[i];

for (long long int i = 0; i < n; i++)

mas[i] = mas\_tmp[i];

}