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»**

**Выполнила**:

студентка группы 3822Б1ПМ1

Вихрева В.В.

**Проверил**:

преподаватель каф. МОСТ,

Волокитин В.Д.

Нижний Новгород

2022

**Содержание**

[Постановка задачи 3](#_Toc26962562)

[Метод решения 4](#_Toc26962563)

[Руководство пользователя 6](#_Toc26962564)

[Описание программной реализации 7](#_Toc26962565)

[Подтверждение корректности 9](#_Toc26962566)

[Результаты экспериментов 10](#_Toc26962567)

[Заключение 1](#_Toc26962568)2

[Приложение 1](#_Toc26962569)3

# Постановка задачи

Задача состоит в реализации на языке Си программы для сортировки массива вещественных чисел типа Float различными алгоритмами:

1. Сортировка «Пузырек»
2. Сортировка «Выбором»
3. Сортировка Хоара («Быстрая»)
4. Сортировка Шелла

Необходимо проверить корректность реализации данных сортировок, сравнить скорость их работы и проверить соответствие теоретической оценке выполнения.

# Метод решения

1. Сортировка «Пузырьком»

Суть алгоритма заключается в сравнении двух соседних элементов. Если текущий больше следующего, то меняем их местами, если нет – не трогаем. Таким образом, большие элементы как бы «всплывают» в «конец» массива. Причем, так как последний элемент сравнивать не с чем, то проход идет до предпоследнего элемента (*j < n -1*, где *n* – длина массива). Более того, с каждым проход в «конце» массива становится все больше упорядоченных элементов (за первый проход в конце оказывается максимальный элемент, за второй – перед максимальным встает первый меньший его элемент и так далее; другими словами, массив упорядочивается с конца, в порядке убывания, а количество упорядоченных элементов напрямую зависит от номера прохода), поэтому с каждым новым проходом мы сравниваем элементы идя не просто до *n*, а до *n – i – 1*, где *i* – номер прохода. Так же можно использовать флаг, чтобы знать делались ли перестановки за проход, если нет – то закончить сортировку. Сложность сортировки О(*n2*).

1. Сортировка «Выбором»

Суть алгоритма заключается в поиске минимального элемента во время прохода, мы меняем его местами с тем элементом, с которого начинали проход. По умолчанию выбираем в качестве минимального элемента тот, с которого начинаем проход, соответственно сравнивать элементы начинаем со следующего. Во время первого прохода мы находим минимальный элемент и ставим его на 1-ое место. Второй проход начинаем уже со второго элемента и находим минимальный среди оставшихся, ставим его на 2-ое место, и так далее. Таким образом массив упорядочивается с начала, в порядке возрастания. Сложность сортировки О(*n2*).

1. Сортировка Хоара («Быстрая»)

Суть алгоритма заключается в выборе ведущего элемента и расположении слева от него чисел меньше него, а справа чисел больше или равных. Далее в каждой половине снова выбирается ведущий элемент и выполняются аналогичные действия, и так далее. Каждый ведущий элемент оказывается точно на своем месте, так как количество элементов в массиве конечно и слева от «ведущего» находятся все элементы меньше, а справа – больше. Одна из реализаций данной сортировки – рекурсивная, так как на каждом шаге мы повторяем одни и те же действия. Сложность сортировки О(*n\*log n*), в худшем случае – О(*n2*).

1. Сортировка Шелла

Суть алгоритма заключается в сравнение разделенных на группы элементов последовательности, находящихся друг от друга на некотором расстоянии. Изначально это расстояние step = N/2, где N — общее число элементов. На первом шаге каждая группа включает в себя два элемента расположенных друг от друга на расстоянии N/2; они сравниваются между собой, и, если элемент с большим порядковым номером меньше другого, меняются местами. На последующих шагах также происходят проверка и обмен, но расстояние step уменьшается вдвое (step /= 2), и количество групп, соответственно, уменьшается. Постепенно расстояние между элементами уменьшается, и на step = 1 проход по массиву происходит в последний раз. Сложность сортировки О(*n\*log n*), в худшем случае – О(*n2*).

# Руководство пользователя

При запуске программа просит пользователя ввести количество элементов в массиве для сортировки. После ввода появляется меню с номерами и соответствующими им действиями (рис. 1). Пользователю необходимо ввести число (от 1 до 5) для выполнения соответствующего действия.

При некорректном вводе программа просит ввести номер команды еще раз (рис. 2).

В случае корректного ввода программа выполнит заданную сортировку и выведет в консоль следующие результаты: время, за которое массив был отсортирован и результат проверки массива на отсортированность («*массив отсортирован верно!»* или *«Ошибка сортировки!»*) (рис. 3). После этого можно выбрать другие сортировки, программа будет работать пока вы не нажмете цифру 5. После нажатия цифры 5 программа заканчивает свою работу.

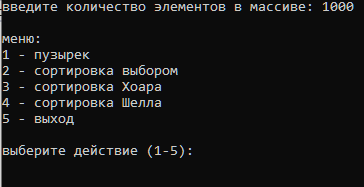
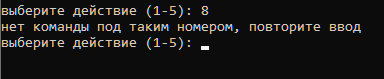
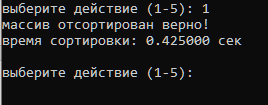


Рис. 2

Рис. 3

Рис. 1

# Описание программной реализации

Программа состоит из одного файла (lab\_No1\_Vikhreva\_V.cpp) в котором расположены все функции: функции сортировок, вспомогательные и главная (main).

**void swap(float\* a, int i, int j)** – функция, меняющая элементы массива местами. Принимает указатель на начало массива, в котором нужно поменять элементы, индексы самих элементов.

**void copy\_arr(float\* arr1, float\* arr2, int n) –** функция копирует один массив в другой. Принимает два указателя на начала массивов (оригинальный и тот, куда копирует) и длину массивов (т.к. она должны быть одинаковой, то и переменная одна).

**int compare(const void\* a, const void\* b)** – функция принимает на вход два указателя на сравниваемые числа и возвращает их взаимное отношение, больше(1)/равно(0)/меньше(-1). Эта функция является вспомогательной для *qsort()* из стандартной библиотеки Си.

**int check(float\* arr, float\* orig\_arr, int na) –** функция проверки корректности работы сортировки (более подробно см. следующий раздел)

**void check\_result(float\* arr, float\* orig\_arr, int na, clock\_t st, clock\_t f) –** функция выводит результат сортировки. Принимает два указателя на начала массивов (отсортированный и неотсортированный). С помощью check проверяет верно ли отсортирован массив. Если верно, то выводит время сортировки и надпись «*массив отсортирован верно!»*, в противно случае выводит *«Ошибка сортировки!».*

**void bubble(float\* arr, int na)** – сортировка «Пузырек». Принимает указатель на начало сортируемого массива и его длину. Первый цикл считает «количество проходов по массиву», оно не должно превышать его размер, вложенный же цикл проходит по массиву и меняет с помощью swap неупорядоченные элементы.

**void choice(float\* arr, int na)** – сортировка «Выбором». Принимает указатель на начало сортируемого массива и его длину. В первом цикле запоминается индекс текущего элемента, как индекс минимального; во вложенном ищется минимальный элемент, если такой находится, то уже его индекс запоминается, как индекс минимального; после этого уже в первом цикле начальный элемент меняется местами с минимальным.

**void QuickSort(float\* arr, int left, int right) –** функция сортировки Хоара(« быстрой»). Принимает указатель на начало сортируемого массива, его левую и правую границы. Осуществляет рекурсию, разделяя данный массив пополам и вызывая себя же для каждой части, пока принимаемый массив не разделит себя до длины 1. Для разделения используется вспомогательная функция *partition*.

**int partition(float\* arr, int left, int right) –** вспомогательная функция к *QuickSort*. Принимает указатель на начало массива, ее левую и правую границы. Находит средний по расположению элемент (*mid*). Начиная от левой границы (*left*) ищет первый элемент больше него, начиная от правой границы (*right*) – первый элемент меньше, при условии, что индекс левого элемента (*left*) меньше индекса правого (*right*), если левый элемент больше правого, то меняет их местами, если нет – уменьшает правый индекс на 1 (смотрит следующий элемент). Это происходит пока левый индекс меньше правого. Функция возвращает значение *right*. Это и будет точка разбиения массива.

**void ShellaSort(float\* arr, int na)** – функция сортировки Шелла. Принимает указатель на начало сортируемого массива и его длину. Основной цикл меняет шаг прохода по массиву (*step*); первый вложенный обеспечивает проход по массиву, начиная с элемента под индексом *i = step*; во втором вложенном цикле идет сравнение (начиная с *j = i - step*) текущего элемента *j* и следующего *j + step*, если текущий больше, они меняются местами.

# Подтверждение корректности

Для подтверждения корректности в программе используется функция:

**int check(float\* arr, float\* orig\_arr, int na)**. Функция принимает указатель на начало отсортированного массива, указатель на начало его неотсортированного оригинала и длину массивов. С помощью *copy\_arr* создает копию оригинального массива и сортирует ее быстрой библиотечной (следовательно, точно верно работающей) сортировкой *qsort.* Затем идет поэлементное сравнение отсортированной копии и массива на проверке, как только нашлись несовпадающие элементы функция возвращает *0*, если же таких нет – *1*.

# Результаты экспериментов

По данным экспериментов видно (рис. 4), что медленнее всего работает сортировка «Пузырек», быстрее нее, но все еще недостаточно работает сортировка «Выбором». До N=10 млн. самой быстрой сортировкой была сортировка Хоара («быстрая»), однако бри N > 10 млн. быстрее работает сортировка Шелла, что особенно видно при N = 50 млн.

Рис. 4

Чтобы проверить соответствие теоретической оценке выполнения, разделим полученное время на теоретическое: для «Пузырька» и «Выбором» на *n2*, а для «Хоара» и «Шелла» на *n\*log n* и *n\*(log n)^2* соответственно, где *n* – количество элементов.

# Заключение

В ходе лабораторной работы на языке Си была реализована программа для сортировки массива вещественных чисел типа Float различными алгоритмами:

1. Сортировка «Пузырек»
2. Сортировка «Выбором»
3. Сортировка Хоара («Быстрая»)
4. Сортировка Шелла

Была сравнена скорость и эффективность работы этих сортировок. Наименее эффективная сортировка - сортировка «Пузырек», после нее идет сортировка «Выбором». Самыми эффективными оказались сортировки Хоара и Шелла. До N=10 млн. сортировка Хоара была быстрее, однако на массивах с бо́льшим размером лучше работает сортировка Шелла.

# Приложение

void swap(float\* a,int i, int j)

{

float t;

t = a[i]; a[i] = a[j]; a[j] = t;

}

//копирование массива

void copy\_arr(float\* arr1, float\* arr2, int n)

{

int i;

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

arr2[i] = arr1[i];

}

//проверка корректности работы

int check(float\* arr, float\* orig\_arr, int na)

{

int i;

float\* arr1;

arr1 = (float\*)malloc(sizeof(float) \* na);

copy\_arr(orig\_arr, arr1, na);

qsort(arr1, na, sizeof(float), compare);

for (i = 0; i < na-1; i++)

if (arr[i] != arr1[i])

return 0;

return 1;

}

//пузырек

void bubble(float\* arr, int na)

{

int i, j, f;

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

{

f = 0;

for (j = 0; j < na - i - 1; j++)

if (arr[j] > arr[j + 1])

{

swap(arr,j,j+1);

f = 1;

}

if (f == 0)

return;

}

}

//выбором

void choice(float\* arr, int na)

{

int i, j, i\_min;

for (i = 0; i < na - 1; i++)

{

i\_min = i;

for (j = i; j < na; j++)

if (arr[j] < arr[i\_min])

i\_min = j;

swap(arr, i, i\_min);

}

}

int partition(float\* arr, int left, int right)

{

float mid = arr[left + (right - left) / 2];

while (left < right)

{

while (arr[left] < mid && (left != right))

left++;

while (arr[right] > mid && (right != left))

right--;

if (left < right) {

if (arr[left] > arr[right])

swap(arr,left,right);

else if (arr[left] == arr[right])

right--;

}

}

return right;

}

//сортировка Хоара

void QuickSort(float\* arr, int left, int right) {

int p;

if (left <= right)

{

p = partition(arr, left, right);

QuickSort(arr, left, p - 1);

QuickSort(arr, p + 1, right);

}

}

//сортировка Шелла

void ShellaSort(float\* arr, int na)

{

int step, i, j;

for (step = na / 2; step > 0; step /= 2)

for (i = step; i < na; i++)

for (j = i - step; j >= 0 && arr[j] > arr[j + step]; j -= step)

swap(arr, j, j + step);

}