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

Порошин В.С.

**Проверил**:

преподаватель каф. МОСТ,

Волокитин В.Д.

Нижний Новгород

2022

**Содержание**

[Постановка задачи 3](#_Toc26962562)

[Метод решения 4](#_Toc26962563)

[Руководство пользователя](#_Toc26962564) 7

[Описание программной реализации](#_Toc26962565) 9

[Подтверждение корректности](#_Toc26962566) 10

[Результаты экспериментов](#_Toc26962567) 11

[Заключение](#_Toc26962568) 12

[Приложение](#_Toc26962569) 15

# Постановка задачи

Описание задачи, что от вас требовалось сделать

Цель – изучение алгоритмов сортировки, работающих за сложность , и .

Для достижения данной цели необходимо поставить следующие задачи:

1. Подготовка тестовой программы, которая умеет изменять размер заданного массива, переключаться между сортировками и выводить корректный ответ;
2. Изучить алгоритмы упорядочивания массива вещественных (типа float) знаковых чисел;
3. Сравнить скорости и эффективности данных сортировок, проверить, что их сложность действительно соответствует теоретической оценке.

# Метод решения

Описание алгоритмов ваших сортировок. Достаточно подробное

Для сравнения я выбрал данные сортировки:

1. Сортировка пузырьком (сложность )
2. Сортировка Шелла (сложность )
3. LSD сортировка (сложность )

Далее предоставляю подробное описание каждой сортировки.

Сортировка пузырьком:

Основная идея состоит в том, что мы выбираем 1-й элемент и сравниваем с его соседом справа, затем меняем их местами, либо не меняем, это зависит от того, как именно вы хотите упорядочить массив, по какому принципу (возрастанию/убыванию), для определенности будем упорядочивать по возрастанию. Итак, на следующем шаге мы должны сравнить 2-й элемент с его соседом справа, затем поменять их местами, если нужно. Таким образом, за 1 цикл (на 1-м шаге n-1 сравнений) самый большой элемент встанет на свое место, т.е. в конец массива. Теперь мы должны выбрать снова 1-й элемент и сравнить с его соседом справа, через n-2 сравнений элемент займет свое место, но надо понимать, что с последним элементом сравнивать не нужно, ведь мы уже выяснили, что он занял свое место, следовательно, количество сравнений будет каждый раз уменьшаться на 1, а мы будем отсортировывать массив «с конца». Проделав n-1 раз описанный выше цикл, можно утверждать, что массив отсортирован. Действительно, чтобы найти место предпоследнему неотсортированному элементу в массиве понадобится ровно 1 сравнение (всего n элементов, предпоследний n-1 элемент встанет на свое место через n-n+1 = 1 сравнений), тогда последний элемент вообще не нужно ни с кем сравнивать, он уже занял свое место (это очевидно, ведь только одно место и осталось) => мы выполнили примерно (n-1)\*(n-1) ~ операций (при достаточно больших n оценка точнее), отсюда мы получаем асимптотику алгоритма О(). Важное замечание! Если установить «флаг», показывающий количество совершенных перестановок элементов за цикл, можно заметить, что как только он станет равен нулю – сортировка окончена. Действительно, так как каждый элемент сравнивается с каждым, то перестановки не будет только тогда, когда он стоит на своем месте, иначе он поменяется хотя бы с одним элементом. А значит, мы можем заканчивать сортировку досрочно, не совершая бесполезных сравнений.

Сортировка Шелла:

Это более быстрая сортировка, основанная на принципе сортировки вставками. Основная идея – ставить сразу несколько элементов «почти» на свое место. В начале мы делим массив пополам (если не делится – округляем вниз), выбираем элемент, индекс которого равен «середине», затем сравниваем с элементом, имеющим индекс = «середина»-шаг, но т.к. шаг = «середине» (на 1 цикле), то это будет 1-й элемент (фактически 0-й, т.к. в С нумерация индексов с 0). Аналогично пузырьку, меняем местами эти элементы, если нужно. Далее выбираем следующий элемент (его индекс = «середина»+1) и сравниваем со 2 элементом (его индекс = середина – шаг + 1 = 1, но это 2-й элемент), если нужно – меняем. Проделываем ту же операцию, пока не дойдем до конца массива (т.е. следующий индекс элемента будет выходить за границы массива), тогда делим нашу «середину» пополам, как в самом начале, а затем повторяем 1-й цикл. На этот раз выбираем элемент, индекс которого равен «четверти» и сравниваем с элементом, индекс которого = «четверть» - шаг (теперь шаг = «четверти»), меняем их по необходимости. Важно отметить, что, начиная с «середины» массива, придется сравнивать уже не 2 элемента, а 3, сначала «середину» с «четвертью», а затем, выполнив или не выполнив перестановку, с 1-м элементом в массиве, если нужно – необходимо поменять его с «четвертью». Таким образом, меняя каждый раз шаг по определенному правилу и переставляя элементы внутри массива, мы отсортируем его за время = . Не трудно заметить, что мы уже не сравниваем каждый элемент с каждым, а выбираем последовательность так, чтобы как можно быстрее поставить элемент на свое место, ведь за 1 сравнение мы можем сдвинуть элемент больше, чем на одну позицию. Очевидно, время работы будет зависеть напрямую от выбранной нами последовательности, в данном случае рассмотрена последовательность Шелла, она не является самой оптимальной, но в ней каждый элемент сравнивается с других элементов, при больших n оценка более точная, но даже при относительно небольшом количестве элементов алгоритм работает за время => его асимптотика .

Линейная сортировка (LSD):

Эта достаточно нетривиальная сортировка, использующая крайне простую идею. В алгоритме не сравниваются элементы между собой, а учитываются особенности сравниваемых типов данных, т.е. сортировка работает только в том случае, если мы знаем, что именно мы сортируем (int, float, double, char и т.д.). Т.к. речь идет о числах, то в качестве примера рассмотрим тип int. Пусть при записи чисел этого типа используется максимум m разрядов (сортировка работает, т.к. компьютер ограничивает числа, он не может хранить в памяти бесконечные числа), тогда выделим последний разряд и переставим числа так, чтобы они располагались по возрастанию этого разряда. Подсчитаем, сколько каких цифр (имеется в виду идея сортировки подсчетом, при этом всего 10 цифр: от 0 до 9), и найдем их места, затем проведем взаимно однозначное соответствие между разрядом и числом, переставим их как было описано выше. Перейдем к следующему разряду (а именно предпоследнему) и также переставим еще раз элементы массива, но уже по возрастанию нового разряда. Повторив эту операцию m раз, мы получим отсортированный массив. Так происходит потому, что, начиная с конца, мы учитываем отличие чисел только в последнем разряде, упорядочиваем их, затем переходим к следующему разряду, если выясняется, что он не одинаков у всех чисел, то переставив по возрастанию числа так, чтобы цифры в этом разряде также шли по возрастанию, мы не нарушаем упорядоченности, достигнутой ранее, ведь если следующий разряд больше, то уже не важно, какие цифры идут после него, пример: 210 > 199. В худшем случае мы сделаем m\*n операций (перестановок), отсюда и асимптотика алгоритма = . Так как все типы данных кодируются байтами, то даже числа с плавающей запятой можно сортировать подобным образом, понимая, как они устроены и записаны в памяти компьютера. Рассмотрим тип float, он кодируется 4-мя байтами и записывается в память в обратном («перевернутом») порядке. Прямой порядок такой: 1 бит знак (если «+» = 0, если «-» = 1), 8 бит порядок (экспонента, она же степень), 23 бит – мантисса (само число). Отсюда не трудно заметить, что если работать с мантиссой, то мы получаем те же утверждения, что и для int, порядок отвечает за степень числа, если у одного числа он больше, чем у другого, то нет смысла рассматривать их мантиссы, пример 1e10 и 1.9е2, единственный нюанс – 1-й байт, в нем учитывается знак, т.е. все отрицательные числа в такой записи > положительных. При сортировке чисел по возрастанию выяснится, что так будут отсортированы только положительные числа, а отрицательные будут расположены после положительных, причем еще и по убыванию. Поэтому после сортировки необходимо учесть этот нюанс и переставить правильно числа, но это уже нетрудная задача. Т.к. float хранится в памяти в обратном порядке, то в алгоритме необходимо выполнять все те же операции, как в случае с int, но начиная с 1-го разряда, а не последнего. В качестве 1-го разряда возьмем 1 байт, для этого запишем float поразрядно через char и установим взаимно однозначное соответствие между числом float и разрядом char, а дальше работаем как с типом int.

# Руководство пользователя

Описание работы с вашей программы, что должен сделать пользователь.

При запуске программы, пользователя попросят ввести количество элементов массива (для того, чтобы пользователь не смог запустить сортировку на пустом массиве).



Рисунок 1 - Что встречает пользователя

Далее пользователь вводит n и начинается основной код программы:

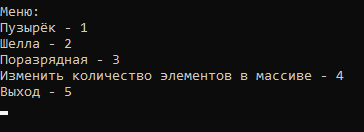


Рисунок 2 - Меню

В данном меню всего 5 ветвлений:

1, 2, 3 клавиши отвечают за запуск соответствующей сортировки.

При запуске программа выводит результат (если программа не сработала и массив не отсортировался, то программа выведет «Ошибка, не удалось отсортировать массив», иначе выведет «Массив отсортирован успешно»). Если массив отсортирован успешно, то программа покажет, за какое время данная сортировка это проделала.

4 клавиша отвечает за перезапись массива (можно поменять как элементы, так и их количество)

5 клавиша отвечает за выход из программы. Он выглядит вот так:



Рисунок 3 - Выход из программы

После выхода невозможно что-либо ввести в терминал.

Отметить можно и то, что данная программа будет работать, пока пользователь не нажмет на клавишу 5, то есть она не завершит свою работу, пока человек сам не скажет, что можно заканчивать работу.

Если пользователь нажмет любую другую клавишу, то программа просто еще раз выведет меню (работает на любой клавише).

# Описание программной реализации

Структура проекта, какие есть файлы и что в них содержится.

Проект состоит из 10 функций, одна из которых main.

void swap(float\* a, float\* b):

функция принимает на вход 2 указателя типа float и меняет местами значения a и b

void copy\_(float\* mas\_a, float\* mas\_b, int size):

функция принимает на вход 2 указателя на массив типа float и размер массива типа int, записывает элементы массива mas\_a в массив mas\_b

int compare(const void\* a, const void\* b):

функция принимает на вход 2 указателя типа const void, сравнивает \*a и \*b, возвращает значения: -1, 0, 1, в зависимости от правила, по которому нужно сортировать (вспомогательная функция)

int check(float\* mas\_a, float\* mas\_b, int size):

функция принимает на вход 2 указателя на массив типа float и размер массива типа int, сравнивает 2 массива одинаковой длины n. Один отсортирован встроенной функцией qsort, а другой - предложенным выше способом.

mas\_a – массив, отсортированный qsort-ом.

mas\_b – массив, отсортированный одним из предложенных выше способов.

void sign(float\* input, int size):

функция принимает на вход указатель на массив типа float, размер массива типа int, переставляет отрицательные и положительные числа в массиве, является вспомогательной функцией для сортировки LSD, не выполняет перестановки, если в исходном массиве отсутствуют отрицательные числа.

void radix(float\* input, int size):

функция принимает на вход указатель на массив типа float, размер массива типа int, сортирует массив методом LSD, описанным выше.

void shell(float\* input, int size, int\* k\_p, int\* k\_s):

функция принимает на вход 3 указателя: один - на массив типа float, два других – на счетчик перестановок и сравнений типа int, размер массива типа int, сортирует массив методом Шелла, описанным выше, считает количество сделанных перестановок и сравнений.

void bubble(float\* input, int size, int\*k\_p, int\*k\_s):

функция принимает на вход 3 указателя: один - на массив типа float, два других – на счетчик перестановок и сравнений типа int, размер массива типа int, сортирует массив методом Пузырька, описанным выше, считает количество сделанных перестановок и сравнений.

void general(float\* input, int size):

функция принимает на вход указатель на массив типа float, размер массива типа int, заполняет массив псевдослучайными числами так, чтобы не получились не числа (NaN), также учтен случай получения «-0», т.к. это альтернативный вид «0», то в генераторе исключена возможность получить такое значение.

int main():

основная функция, ничего не принимает и не возвращает (фактически возвращает всегда 0), она отвечает за корректную работу программы, перенаправление в нужные подпрограммы и взаимодействие с пользователем.

# Подтверждение корректности

Для подтверждения корректности в программе используется функция check. Она принимает на вход 2 массива, один из которых является отсортированным одним из методов, предложенных выше, второй же отсортирован встроенной функцией qsort. qsort принимает 2-й массив, который является неотсортированным, тип данных его элементов – float, также для корректной работы требуется специальная функция (компоратор), которая является критерием сортировки. Далее идет сравнение поэлементно. Если хотя бы 1 элемент на месте i в обоих массивах не совпал, то функция возвращает 0. Иначе вернется 1, а значит массивы равны.

# Результаты экспериментов

По данным экспериментов видно, что сортировка пузырьком показала самый плохой результат. Чем больше элементов, тем больше времени она тратит на сортировку массива, ее график вычерчивает одну из ветвей параболы (не трудно заметить данное соответствие). Это неудивительно, ведь у нее самая плохая сложность .

Значительно быстрее работает сортировка Шелла. Она начала показывать время с 5 миллионов элементов, сортировка заняла 6 секунд. Далее можно заметить рост времени у сортировки поразрядной. Она оторвалась от точки 0 на 20 миллионах элементов, затратив 1 секунду на сортировку всего массива.

Для тестирования сортировок использовались массивы размером

1. 1000 элементов
2. 10 000 элементов
3. 25 000 элементов
4. 50 000 элементов
5. 100 000 элементов
6. 1 000 000 элементов
7. 5 000 000 элементов
8. 10 000 000 элементов
9. 20 000 000 элементов
10. 50 000 000 элементов

На основе этих данных была построена диаграмма

# Заключение

Из тестов мы можем сказать, что самая эффективная сортировка – это сортировка поразрядная. Она показала самый хороший результат. Начала рост с 20 миллионов элементов и затратила 1 секунду на полную сортировку. Далее по эффективности идет сортировка Шелла, а на последнем месте сортировка пузырьком. (Уточнение: в данном эксперименте учитывалось только целое значение времени в секундах.)

Для реализации данной работы мы создали свою систему тестирования, изучили и написали алгоритмы сортировки, а также экспериментально подтвердили теоретическую оценку сложности сортировок. (Уточнение: для подтверждения мы делили время в микросекундах (т.к. в секундах значения слишком малы) на заявленную теорией сложность алгоритма, если с какого-либо момента все последующие значения мало отличались друг от друга и были примерно равны константе, то оценка подтверждалась. Как видно из графика: для пузырька C ~ 0,005; для сортировки Шелла C ~ 0,03; для сортировки поразрядной C ~ 0,055. Отсюда делаем вывод, что теоретическая сложность соответствует действительности.)

# Приложение

void swap(float\* a, float\* b) {

float tmp = \*a;

\*a = \*b;

\*b = tmp;

}

void sign(float\* input, int size) {

float\* output;

int key, i, j;

key = -1;

output = (float\*)malloc(size \* sizeof(float));

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

if (input[i] < 0) {

key = i;

break;

}

if (key != -1) {

for (j = size - 1; j >= key; j--)

output[size - j - 1] = input[j];

for (j = 0; j < key; j++)

output[j + size - key] = input[j];

for (j = 0; j < size; j++)

input[j] = output[j];

}

}

void radix(float\* input, int size) {

int i, j;

int step = 4;

int number;

unsigned char\* mas;

float\* output;

int count[256];

int sum[256];

for (number = 0; number < step; number++) {

mas = (unsigned char\*)input;

output = (float\*)malloc(size \* sizeof(float));

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

count[i] = 0;

sum[i] = 0;

}

for (i = number; i < step \* size; i += step)

count[mas[i]]++;

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

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

if (count[i] != 0)

sum[i] += count[j];

for (i = number; i < step \* size; i += step) {

output[sum[mas[i]]] = input[i / step];

sum[mas[i]]++;

}

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

input[i] = output[i];

}

sign(input, size);

}

void shell(float\* input, int size, long long int\* k\_p, long long int\* k\_s) {

int i, j, k;

for (i = size / 2; i > 0; i /= 2)

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

for (k = j - i; (k > -1) && (++\*k\_s && (input[k] > input[k + i])); k -= i) {

swap(&input[k], &input[k + i]);

++\*k\_p;

}

}

void bubble(float\* input, int size, long long int\*k\_p, long long int\*k\_s) {

int i, j, swapped;

for (i = 0; i < size - 1; i++) {

swapped = 0;

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

if (++\*k\_s && (input[j] > input[j + 1])) {

swap(&input[j], &input[j + 1]);

++\*k\_p;

swapped = 1;

}

}

if (swapped == 0) break;

}

}

void general(float\* input, int size)

{

int i;

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

input[i] = ((float)rand()) /(0.01 + (float)(rand()));

if (rand() % 2 == 0 && input[i] != 0)

input[i] = -input[i];

}

}