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/ка группы 3822Б1ПМ1

Волочаев. С. В.

**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

Описание задачи, что от вас требовалось сделать

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

Для выполнения 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 double) знаковых чисел;
3. Сравнить скорости и эффективности данных сортировок.

# Метод решения

Описание алгоритмов ваших сортировок. Достаточно подробное

Для сравнения я выбрал данные сортировки:

1. Сортировка выбором (сложность )
2. Сортировка расческой (сложность )
3. LSD сортировка (сложность )

Далее предоставляю подробное описание каждой сортировки.

Сортировка выбором.

Алгоритм данной сортировки заключается в том, чтобы найти наименьший элемент в оставшемся массиве и поставить его в начало.

Алгоритм проходится по массиву раз для обнаружения наименьшего из оставшихся элементов и ставит его в начало подмассива. Под подмассивом подразумевается массив, который начинается с i-ого элемента. Когда мы ищем наименьший элемент, то мы запоминаем начало массива, и предполагаем, что он наибольший. Далее мы проходимся по подмассиву до конца и если оказывается, что в нем присутствует элемент, который меньше начала, то мы запоминаем его индекс. Если после этого мы находим элемент еще меньше, то мы запоминаем его индекс и так далее до конца подмассива. Последним шагом в каждом подмассиве мы меняем местами элементы с начала массива и элемент, который был минимальный. Далее длина подмассива уменьшается на 1 (мы убираем 1 элемент слева). Таким образом мы разделяем массив на 2 части. Одна является отсортированной (левая), а другая оставшийся подмассив.

Данная сортировка будет продолжаться до тех пор, пока длина подмассива не станет равна 1.

Сортировка расческой.

Данный алгоритм является улучшением сортировки «Пузырек». Под улучшением подразумевается то, что мы будем сравнивать элементы не соседние, а через какой-то шаг. Для этого мы заведем определенную переменную, которая будет отвечать за изменение шага. В начале шаг равен размеру массива. Пока шаг не будет равен 0 (а это точно когда-нибудь произойдет) мы делим его на эту переменную и запускаем цикл по элементам и сравниваем их через этот шаг. Если элемент, который стоит ближе к началу больше чем элемент, который стоит дальше от начала, то мы их меняем местами. Так будет продолжаться пока шаг не станет равным 0. Это и будет означать, что массив отсортирован. Так как мы сравниваем элементы через какой-то шаг, то сложность сортировка уменьшается. Ее сложность в итоге получится

LSD сортировка

У данной сортировки сложность . Это происходит из-за того, что мы не сравниваем числа, а используем их байты. Мы делим число на 8 частей (по 1 байту каждый), проходимся поочередно по каждому байту (от конца до начала) и каждый раз используем побитовую сортировку. Таким образом достигается максимально возможная скорость сортировки.

Немного про побитовую сортировку.

Она представляет собой массив из 256 элементов, в котором лежит количество чисел, с данными байтами. Далее мы расставляем их в массиве так, чтобы байты в массиве соответствовали их числам.

# Руководство пользователя

Описание работы с вашей программы, что должен сделать пользователь.

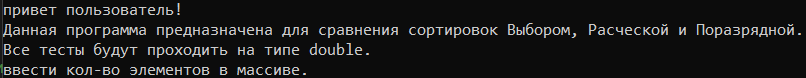
Когда запускаешь программу сразу на вход она принимает кол-во элементов массива (для того, чтобы пользователь не смог запустить сортировку на пустом массиве).

Рисунок что встречает пользователя

Далее пользователь вводит n и начинается основной код программы:

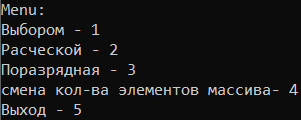


Рисунок Меню

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, массив не отсортировался», иначе выведет «Массив отсортирован успешно»). Если массив отсортирован успешно, то программа введет 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 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 не упадет, а 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 11 функций, одна из которых main.

Также для упрощения написания кода были введены некоторые замены типов данных:

ll – long long int.

uchar – unsigned char.

функции:

void swap(double\* a, double\* b)

данная функция необходима для замены элементов друг на друга.

а и b - элементы, которые необходимо поменять местами.

void reverse(double\* mas, ll n, ll k, ll\* kol\_permutation)

данная функция отвечает за переворот массива.

mas – сам массив.

n – количество элементов в массиве.

k - сколько элементов не надо переворачивать (с конца).

kol\_permutation – переменная для подсчета перестановок (указатель на память, так как она будет возвращаться в исходную программу)

void filling(double\* mas, ll n, ll\* kol\_negative)

данная функция отвечает за заполнение массива случайными числами (отрицательные тоже учтены).

mas – массив.

n – количество элементов.

kol\_negative – количество отрицательных элементов, которые будут в массиве. Программа сама вернет данное число, так как оно передается через указатель.

void copy\_(double\* mas\_a, double\* mas\_b)

данная функция копирует mas\_a в mas\_b.

mas\_a – массив а.

mas\_b – массив b.

на выходе из функции получится 2 массива а.

void choice\_sort(double\* mas, ll n, ll\* kol\_comparison, ll\* kol\_permutation)

Реализация сортировки выбором.

mas – массив.

n – размер массива.

kol\_comparison – переменная для подсчета количества сравнений (указатель на память, так как она будет возвращаться в исходную программу).

kol\_permutation – переменная для подсчета перестановок (указатель на память, так как она будет возвращаться в исходную программу)

void comb\_sort(double\* mas, ll n, ll\* kol\_comparison, ll\* kol\_permutation)

реализация сортировки расческой.

mas – массив.

n – размер массива.

kol\_comparison – переменная для подсчета количества сравнений (указатель на память, так как она будет возвращаться в исходную программу).

kol\_permutation – переменная для подсчета перестановок (указатель на память, так как она будет возвращаться в исходную программу)

void radix\_sort(double\* mas, double\* out, ll byte, ll n, ll\* kol\_permutation)

реализация побитовой сортировки.

mas – исходный массив.

out – массив, который пойдет в ответ.

byte – номер байта, по которому сортируем.

n – количество элементов.

kol\_permutation – переменная для подсчета перестановок (указатель на память, так как она будет возвращаться в исходную программу)

void LSD(double\* mas, ll n, ll k, ll\* kol\_permutation)

данная функция идет в дополнение к функции побитовой сортировки для расширения ее на тип double.

mas – массив.

n – количество элементов.

k – количество отрицательных элементов.

kol\_permutation – переменная для подсчета перестановок (указатель на память, так как она будет возвращаться в исходную программу)

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

дополнительная функция (компоратор) для проверки правильности сортировки. Она сравнивает 2 числа.

\*a – первый элемент для сравнения.

\*b – второй элемент для сравнения.

ll check(double\* mas\_a, double\* mas\_b, ll n)

данная функция сравнивает 2 массива одинаковой длины n. Один отсортирован встроенной функцией qsort, в другая своей реализацией.

mas\_a – массив, отсортированный qsort-ом.

mas\_b – массив, отсортированный своей реализацией.

int main() – основная функция

# Подтверждение корректности

Для подтверждения корректности в программе используется функция check. В нее подается 2 массива, один из которых является отсортированный собственной реализацией. Второй же я решил отсортировал встроенной функцией qsort. В нее был подан 2 массив. Который является не отсортированным, тип данных, в котором представлены его элементы и 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. Если не произошел 0, то вернется 1, а значит 2 массива равны.

# Результаты экспериментов

Для тестирования сортировок я использовал массивы размером

1. 100 000 элементов
2. 200 000 элементов
3. 400 000 элементов
4. 800 000 элементов
5. 1 000 000 элементов
6. 20 000 000 элементов
7. 50 000 000 элементов
8. 100 000 000 элементов
9. 250 000 000 элементов

500 000 000 элементов увы не уместились в оперативную память.

Получился такой график:

По выше составленному графику можно сказать, что самая быстрая сортировка – поразрядная, а самая медленная – выбором. Из этого можно сделать вывод, что у сортировки выбором должна быть сложность , а у поразрядной . Это необходимо доказать.

Для того, чтобы подтвердить сложность алгоритма необходимо доказать, что константа сложности (которую мы отбрасываем при определении алгоритма) именно константа по сравнению с самой сложностью. Для этого необходимо разделить время, за которое обработалось данное количество элементов массива, на сложность самого алгоритма с подстановкой самого n.

Сортировка выбором.

У данной сортировки сложность . Докажем это приведенным выше способом.

Из диаграммы видно, что при увеличении количества элементов функция выходит на константу, а это означает, что сложность сортировки доказана.

Сортировка расческой.

У данной сортировки сложность . Но основание не определено, так как в сложности оно не играет значения. Для определенности скажем, что оно равно 1.247(так как мы использовали именно такой шаг).

Так же можно увидеть выход на константу, а значит данная сортировка соответствует поставленной сложности.

Поразрядная сортировка.

У данной сортировки сложность .

Тоже в итоге выходит на константу, а значит соответствует поставленной сложности.

# 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 показала 2 секунды на полную сортировку. Далее по эффективности идет сортировка расческой, и на последнем месте сортировка выбором.

Для реализации данной работы мы создали свою систему тестирования, изучили и написали алгоритмы сортировки.

# Приложение

Алгоритм поразрядной сортировки

void radix\_sort(double\* mas,double\* out, ll byte, ll n)

{

uchar\* masc = (uchar\*)mas;

ll counter[256];

ll tmp;

ll i;

for (i = 0; i < 256; ++i) counter[i] = 0;// заполнение нулями, как количество байт

for (i = 0; i < n; ++i) counter[masc[8 \* i + byte]]++;//заполнение массива количеством байт

tmp = counter[0];

counter[0] = 0;

for (i = 1; i < 256; ++i)

{

ll b = counter[i];

counter[i] = tmp;

tmp += b;

}

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

{

out[counter[masc[8 \* i + byte]]] = mas[i];

counter[masc[8 \* i + byte]]++;

}

}

void LSD(double\* mas, ll n,ll k)

{

ll i;

double\* out = (double\*)malloc(n \* sizeof(double));

for (i = 0; i < 8; ++i)

{

(i % 2 == 0) ? radix\_sort(mas, out, i, n) : radix\_sort(out, mas, i, n);

}

reverse(mas, n, k);

reverse(mas, n, 0);

}

Алгоритм сортировки расческой

void comb\_sort(double\* mas, ll n)

{

double factor = 1.247;

double step = n / factor;

while (step - 1 > 0)

{

ll i;

ll st = step;

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

{

if (mas[i] > mas[i + st])

swap(&mas[i], &mas[i + st]);

}

step /= factor;

}

}

Алгоритм сортировки выбором

void choice\_sort(double\* mas, ll n)

{

ll i;

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

{

ll min = i;

ll j;

for (j = i + 1; j < n; ++j)

{

if (mas[j]-mas[min] < 0) min = j;

}

if (i != min) swap(&mas[i], &mas[min]);

}

}