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

Зазнобин П.В.

**Проверил**:

преподаватель каф. МОСТ,

Волокитин В.Д.

Нижний Новгород

2022

**Содержание**

[Постановка задачи 3](#_Toc123230380)

[Метод решения 4](#_Toc123230381)

[Руководство пользователя 6](#_Toc123230382)

[Описание программной реализации 8](#_Toc123230383)

[Подтверждение корректности 12](#_Toc123230384)

[Результаты экспериментов 13](#_Toc123230385)

[Заключение 16](#_Toc123230386)

[Приложение 17](#_Toc123230387)

# Постановка задачи

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

Для выполнения данной цели необходимо поставить и выполнить следующие задачи:

1. Подготовка тестовой программы, которая имеет пользовательский интерфейс, умеет считать время работы отдельных функций и сравнивает результат работы созданных сортировок с результатом заведомо корректной функции.
2. Изучить алгоритмы упорядочивания массива вещественных (типа double) знаковых чисел;
3. Сравнить скорости и эффективности данных сортировок.

# Метод решения

Мне были предоставлены для реализации следующие сортировки:

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

**Сортировка пузырьком.**

На каждом шаге мы находим наибольший элемент из двух соседних и ставим этот элемент в конец пары. Получается, что при каждом прогоне цикла большие элементы будут всплывать к концу массива, как пузырьки воздуха.

Алгоритм выглядит так:

1. Берём первый элемент массива и сравниваем его со вторым. Если первый больше второго — меняем их местами с первым, если нет, идем дальше
2. Берём второй элемент массива и сравниваем его со следующим. Если второй больше— меняем их местами, если нет, идем дальше.
3. Проходим так до предпоследнего элемента, сравниваем его с последним и ставим наибольший из них в конец массива. Всё, мы нашли самое большое число в массиве и поставили его на своё место.
4. Возвращаемся в начало алгоритма и делаем всё снова точно так же, начиная с первого элемента. Только теперь идем до элемента, который находится перед предпоследним, потому что на последнем месте уже стоит самый большой элемент.
5. Повторяем до тех пор, пока у нас не останется один элемент.

В своем проекте я использовал немного видоизмененную версию пузырька. Я использовал флаг, для того чтобы в ситуации, когда я еще не «поднял наверх» все элементы массива, но уже знаю, что массив полностью отсортирован, я мог прекратить сортировку.

Таким образом, в худшем случае мы идем от нулевого элемента до n-1 – ого элемента

**Сортировка слиянием.**

Сортировка состоит из двух функций. Первая функция (merge\_sort) делит, в моем случае рекурсивно, массив каждый раз пополам, пока не дойдет до отдельно стоящих элементов и отправляет сначала по два элемента, чтобы слить их в отсортированном порядке в отдельный массив, потом по 4 элемента, каждый раз умножая количество отправляемых элементов в два раза, пока массив не закончится.

Вторая функция (merge) занимается непосредственно слиянием двух массивов одного размера в один целый отсортированный вспомогательный массив.

В конце всей сортировки в изначальный массив заносятся все элементы из отсортированного вспомогательного массива.

**Поразрядная сортировка.**

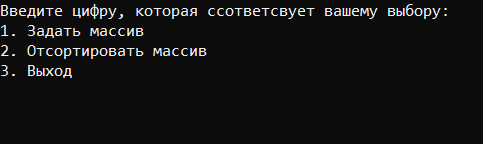
Отличие этой сортировки от других заключается в том, что в ней мы не используем сравнение.

Здесь мы делим каждое отдельное число на байты и каждый раз используем для них побитовую сортировку, не меняя порядок сортируемых элементов, если вдруг значение байта оказалось одинаковым.

Для того, чтобы реализовать побитовую сортировку, мы используем массив из 256 элементов, в котором лежит количество чисел с данными байтами. Потом мы расставляем эти числа в массиве так, чтобы байты в массиве соответствовали их числам.

В конце мы получаем отсортированный массив, но который имеет отрицательные числа в самом конце, при том, что мы сортировали по возрастанию. Это получилось из-за того, что у отрицательных чисел в побитовом представлении на первом месте идет единица. Поэтому нам следует поставить их правильным образом вниз.

# Руководство пользователя



Рисунок

На Рисунке 1 представлено начальное меню. Тут можно выбрать то, что хочется сделать. Однако, если массив еще не задан, рекомендуется его задать, нажав «1»



Рисунок 2

Далее требуется ввести размерность массива, потому что он задается динамически.

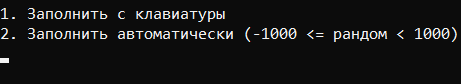


Рисунок 3

Теперь можно ввести числа, которые требуется отсортировать или задать их автоматически с помощью псевдорандома.

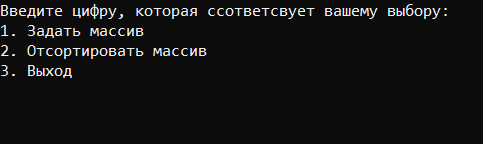


Рисунок 4

Опять попадаем на меню, где можем либо заново задать массив или отсортировать имеющийся, который мы только что был создан и заполнен.

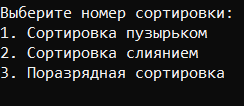


Рисунок 5

На этом этапе можно выбрать тип сортировки.



Рисунок 6

По итогу мы получаем вывод о том, что сортировка прошла успешна и результат корректен, так как схож с результатом сортировки этих же чисел на заведомо корректной встроенной функции.

Также написано время в тиках, за которое была произведена сортировка.

При нажатии «3» в меню, произойдет выход из программы.

При некорректном вводе числа в меню, меню вызовется заново.

При успешной сортировке можно продолжить сортировать тот же массив другим способом или создать новый массив для последующей сортировки. При этом память будет бережно очищена.

# Описание программной реализации

Мой проект состоит из 11 функций, одна из которых функция main().

**Функция меню:**

int menu() {

int v;

printf\_s("1. Задать массив\n2. Отсортировать массив\n3. Выход\n");

scanf\_s("%d", &v);

return v;

}

**Функция swap’а двух переменных в массиве:**

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

float tmp = \*a;

\*a = \*b;

\*b = tmp;

}

**Сортировка пузырьком:**

void bubble\_sort(float\* arr, int n) {

int i, j;

int flag = 0;

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

flag = 0;

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

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

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

flag = 1;

}

}

if (flag == 0)

break;

}

}

**Рекурсивное деление массива пополам до одного элемента включительно и вызов функции слияния и для каждых двух частей разделенных:**

void merge\_sort(float\* arr, float\* tmp, int l, int r) {

int m = l + ((r - l) / 2);

if (l == r) {

return;

}

merge\_sort(arr, tmp, l, m);

merge\_sort(arr, tmp, m + 1, r);

merge(arr, tmp, l, m, r);

}

**Сортировка и слияние двух частей массива:**

void merge(float\* in, float\* out, int l, int m, int r) {

int l1, l2;

int index = 0;

l1 = l;

l2 = m + 1;

while (l1 <= m && l2 <= r) {

if (in[l1] <= in[l2]) {

out[index++] = in[l1++];

}

else {

out[index++] = in[l2++];

}

}

while (l1 > m && l2 <= r) {

out[index++] = in[l2++];

}

while (l2 > r && l1 <= m) {

out[index++] = in[l1++];

}

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

in[l + i] = out[i];

}

}

**Функция, которая сортирует отдельные байты:**

void countByte(unsigned int\* arr, int size, int count[256], int Byte) {

int tmp1, tmp2;

int i;

unsigned char\* arrc = 0;

arrc = (unsigned char\*)malloc(sizeof(unsigned char) \* 4\* size);

int bias;

bias = sizeof(unsigned int);

arrc = (unsigned char\*)arr;

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

count[i] = 0;

}

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

count[arrc[i \* bias + Byte]]++;

}

tmp1 = count[0];

count[0] = 0;

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

tmp2 = count[i];

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

tmp1 = tmp2;

}

**Функция, которая дает на вход предыдущей функции отдельные байты по порядку:**

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

unsigned char\* arrc = (unsigned char\*)arr;

int count[256];

int sizetype = sizeof(unsigned int);

int i, j;

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

countByte(arr, size, count, i);

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

arr\_tmp[count[arrc[j \* sizetype + i]]++] = arr[j];

}

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

arr[j] = arr\_tmp[j];

}

}

}

**Функция, которая позволяет работать с отрицательными числами и числами типа Float:**

void radixFloat(float\* arr, float\* arr\_tmp, int N) {

int count = 0;

int i = 0;

radix\_sort((unsigned int\*)arr, (unsigned int)N, (unsigned int\*)arr\_tmp);

while (i < N) {

if (arr[i] >= 0) {

count++;

}

i++;

}

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

arr\_tmp[N - count + i] = arr[i];

}

count = N - count;

for (count = count--; count >= 0; count--) {

arr\_tmp[count] = arr[i++];

}

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

arr[i] = arr\_tmp[i];

}

}

**Функция, которая позволяет проверить функцию сортировки на корректность, сравнивая ее с заведомо корректной функции:**

int check(float\* arr, float\* arrforsort, int n) {

int f = 0;

int i = 0;

float\* qarr = arr;

qarr = (float\*)malloc(sizeof(float) \* n);

qarr = arr;

qsort(qarr, n, sizeof(float), compare);

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

if (qarr[i] != arrforsort[i]) {

return 0;

}

}

return 1;

}

**Компаратор, который позволяет задать правила сортировки для стандартной функции sort:**

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

{

const float\* ad, \* bd;

ad = (const float\*)a;

bd = (const float\*)b;

if ((\*ad - \*bd) < 0) return -1;

else

{

if ((\*ad - \*bd) > 0) return 1;

else return 0;

}

}

**Основная функция:**

**Main()**

# Подтверждение корректности

Для подтверждения корректности, в проекте используется функция check, которая сравнивает элементы отсортированного массива моими функциями с элементами массива, отсортированного стандартной функцией sort. Функция check возвращает «0», если массив был отсортирован неверно и «1», если результат верен.

# Результаты экспериментов

Для тестирования сортировок я использовал массивы размером

1. 10 000 элементов
2. 30 000 элементов
3. 100 000 элементов
4. 200 000 элементов
5. 400 000 элементов
6. 800 000 элементов
7. 1 000 000 элементов
8. 20 000 000 элементов
9. 50 000 000 элементов
10. 100 000 000 элементов

По выше составленному графику можно сказать, что самая быстрая сортировка – поразрядная, а самая медленная – пузырьком. Из этого можно сделать вывод, что у сортировки сортировки 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 пузырьком:**

Теоретическая асимптотика этого алгоритма составляет примерно ***n^2***.

Попробую это доказать приведенным выше способом.

Из графика видно, что график стремится к константе, значит, можно сказать, что асимптотика верная.

**Сортировка слиянием:**

Теоретическая асимптотика этого алгоритма составляет примерно ***nlog(n)***.

Попробую это доказать приведенным выше способом.

Из графика видно, что график находится вблизи константы, значит, можно сказать, что асимптотика верная.

**Поразрядная сортировка:**

Теоретическая асимптотика этого алгоритма составляет примерно ***n***.

Попробую это доказать приведенным выше способом.

Из графика видно, что график находится вблизи константы, значит, можно сказать, что асимптотика верная.

Во всех доказательствах существует некая маленькая погрешность, из-за применения тиков, которые по своей природе не постоянны.

# Заключение

Из тестов мы можем сказать, что самая эффективная сортировка – поразрядная сортировка. Она меньше всего затрачивает времени на сортировку при больших числах. Следом за ней по скорости работы идет сортировка слиянием и на самом последнем месте с огромным отрывом идет сортировка пузырьком.

Для реализации проекта мной была создана программа, написаны сортировки и было проанализировано их скорость работы на большом диапазоне количества элементов.

# Приложение

#include "stdio.h"

#include "cstdlib"

#include "locale.h"

#include "time.h"

#include "string.h"

int menu() {

int v;

printf\_s("1. Задать массив\n2. Отсортировать массив\n3. Выход\n");

scanf\_s("%d", &v);

return v;

}

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

float tmp = \*a;

\*a = \*b;

\*b = tmp;

}

void bubble\_sort(float\* arr, int n) {

int i, j;

int flag = 0;

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

flag = 0;

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

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

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

flag = 1;

}

}

if (flag == 0)

break;

}

}

void merge(float\* in, float\* out, int l, int m, int r) {

int l1, l2;

int index = 0;

l1 = l;

l2 = m + 1;

while (l1 <= m && l2 <= r) {

if (in[l1] <= in[l2]) {

out[index++] = in[l1++];

}

else {

out[index++] = in[l2++];

}

}

while (l1 > m && l2 <= r) {

out[index++] = in[l2++];

}

while (l2 > r && l1 <= m) {

out[index++] = in[l1++];

}

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

in[l + i] = out[i];

}

}

void merge\_sort(float\* arr, float\* tmp, int l, int r) {

int m = l + ((r - l) / 2);

if (l == r) {

return;

}

merge\_sort(arr, tmp, l, m);

merge\_sort(arr, tmp, m + 1, r);

merge(arr, tmp, l, m, r);

}

void countByte(unsigned int\* arr, int size, int count[256], int Byte) { //исхоодный массив, размер массива, символьный массив, текущий разряд

int tmp1, tmp2; //временная переменная

int i;

unsigned char\* arrc = 0; // символьный массив

arrc = (unsigned char\*)malloc(sizeof(unsigned char) \* 4\* size);

int bias;

bias = sizeof(unsigned int); //он хранит размер беззнакового инта

arrc = (unsigned char\*)arr; //делаю массив символов из предыдущего массива

for (i = 0; i < 256; i++) { //заполняю вспомогательный массив нулями

count[i] = 0;

}

for (i = 0; i < size; i++) { //накидываю сколько раз определенное число входило в массив

count[arrc[i \* bias + Byte]]++;

}

tmp1 = count[0]; //распределяю по индескам куда ставить числа

count[0] = 0;

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

tmp2 = count[i];

count[i] = count[i - 1] + tmp1; //теперь в count лежат индексы, на которые нужно ставить числа каждого индекса массива

tmp1 = tmp2;

}

}

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

unsigned char\* arrc = (unsigned char\*)arr; //копирую массив чаров

int count[256];

int sizetype = sizeof(unsigned int);

int i, j;

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

countByte(arr, size, count, i); //где i - это номер разряда

for (j = 0; j < size; j++) { //Заполняем числами с выведенной индексацией

arr\_tmp[count[arrc[j \* sizetype + i]]++] = arr[j];

}

for (j = 0; j < size; j++) { //Из временного массива в основной

arr[j] = arr\_tmp[j];

}

}

}

void radixFloat(float\* arr, float\* arr\_tmp, int N) {

int count = 0;

int i = 0;

radix\_sort((unsigned int\*)arr, (unsigned int)N, (unsigned int\*)arr\_tmp);

while (i < N) { //Считаем количество положительных чисел

if (arr[i] >= 0) {

count++;

}

i++;

}

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

arr\_tmp[N - count + i] = arr[i];

}

count = N - count;

for (count = count--; count >= 0; count--) {

arr\_tmp[count] = arr[i++];

}

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

arr[i] = arr\_tmp[i];

}

}

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

{

const float\* ad, \* bd;

ad = (const float\*)a;

bd = (const float\*)b;

if ((\*ad - \*bd) < 0) return -1;

else

{

if ((\*ad - \*bd) > 0) return 1;

else return 0;

}

}

int check(float\* arr, float\* arrforsort, int n) {

int f = 0;

int i = 0;

float\* qarr = arr;

qarr = (float\*)malloc(sizeof(float) \* n);

qarr = arr;

qsort(qarr, n, sizeof(float), compare);

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

if (qarr[i] != arrforsort[i]) {

return 0;

}

}

return 1;

}

int main()

{

int v, n, v2, v3, i;

float\* arrforsorts = 0;

float\* arr = 0;

clock\_t start, end;

setlocale(LC\_ALL, "Rus");

while (true) {

int f = 0;

printf("Введите цифру, которая ссответсвует вашему выбору:\n");

v = menu();

switch (v) {

case(1):

{

printf("Укажите размерность массива:\n");

scanf\_s("%d", &n);

arr = (float\*) malloc(sizeof(float) \* n);

arrforsorts = (float\*)malloc(sizeof(float) \* n);

printf("1. Заполнить с клавиатуры\n2. Заполнить автоматически (-1000 <= рандом < 1000)\n");

scanf\_s("%d", &v2);

switch (v2) {

case(1):

{

printf("Введите %d чисел:\n", n);

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

printf("%d. ", i + 1);

scanf\_s("%f", &arr[i]);

arrforsorts[i] = arr[i];

}

continue;

}

case(2):

{

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

arr[i] = (((float) rand() / RAND\_MAX) \* 1000) - (((float)rand() / RAND\_MAX) \* 1000);

arrforsorts[i] = arr[i];

}

continue;

}

default:

{

free(arrforsorts);

free(arr);

continue;

}

}

}

case (2):

{

printf("Выберите номер сортировки:\n1. Сортировка пузырьком\n2. Сортировка слиянием\n3. Поразрядная сортировка\n");

scanf\_s("%d", &v3);

switch(v3) {

case(1):

{

start = clock();

bubble\_sort(arrforsorts , n);

end = clock();

if (check(arr, arrforsorts, n) == 1) {

printf\_s("OK\n");

printf\_s("%d\n", (end - start));

}

else printf\_s("ERORR!\n");

continue;

}

case(2):

{

float\* out = 0;

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

start = clock();

merge\_sort(arrforsorts, out, 0, n-1);

end = clock();

if (check(arr, arrforsorts, n) == 1) {

printf\_s("OK\n");

printf\_s("%d\n", (end - start));

}

else printf\_s("ERORR!\n");

free(out);

continue;

}

case(3):

{

float\* arr\_tmp = (float\*)malloc(sizeof(float) \* n);

start = clock();

radixFloat(arrforsorts, arr\_tmp, n);

end = clock();

if (check(arr, arrforsorts, n) == 1) {

printf\_s("OK\n");

printf\_s("%d\n", (end - start));

}

else printf\_s("ERORR!\n");

free(arr\_tmp);

continue;

}

}

}

case(3):

{

f = 1;

}

default:

{

free(arr);

break;

}

}

if (f == 1) {

break;

}

free(arrforsorts);

free(arr);

}

}