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 382003-1

Ларин К.Д.

**Проверил**:

ассистент каф. МОСТ,

Волокитин В.Д.

Нижний Новгород

2020

**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

Реализовать алгоритмы сортировки массива типа float на языке C: bubble sort, quick sort, merge sort, radix sort, реализовать функцию проверки корректности сортировки, экспериментально подтвердить асимптотическую сложность алгоритмов.

# Метод решения

1. Bubble sort:

Алгоритм состоит из повторяющихся проходов по массиву. При каждом проходе сравниваются соседние элементы, если элементы массива идут в обратном порядке (по возрастанию) они меняются местами. Если при очередном проходе не произошло замен или число выполненных проходов совпадает с размером массива, то массив отсортирован.

1. Quick sort:

Алгоритм состоит из следующих шагов:

* выбрать элемент из массива (используя псевдослучайные числа или другие алгоритмы)
* переставить элементы массива так чтобы сначала шли числа меньше или равные опорного, затем сам опорный элемент, после числа большие или равные ему
* для частей массива с «меньшими» и «большими» значениями применить тот же алгоритм, пока эти части включают в себя хотя бы два элемента.

1. Merge sort:

Алгоритм заключается в том, чтобы разделить массив на две части, отсортировать каждую по отдельности, а после соединить части в один отсортированный массив. Сортировка каждой части выполняется тем же алгоритмом, пока в массиве больше одного элемента, массив из одного элемента — отсортированный. Слияние отсортированных частей выполняется в дополнительный массив по следующему алгоритму:

* пока обе части исходного массива не рассмотрены полностью в конец дополнительный массив добавляется меньший их еще не рассмотренных элементов частей массива
* когда одна часть исходного массива рассмотрена в дополнительный массив копируется оставшаяся часть не рассмотренного массива.

После числа из дополнительного массива по порядку копируются в исходный.

1. Radix sort:

элементы массива сортируются по значению одного из байт их преставления в памяти компьютера, байт выбирается последовательно от младшего к старшему. При каждом проходе осуществляется сортировка подсчетом:

* подсчитывается количество элементов массива со всеми возможными значениями очередного байта
* новая позиция числа со значением байта равным i определяется как количество элементов со значением байта меньшим i
* на полученные позиции записываются числа из исходного массива, после записи позиция увеличивается на 1.

# Руководство пользователя

В первой строке консоли выводятся первые 50 элементов или сообщение, что массив пуст . В следующей строке выводится текущий размер массива.

Работа с программой осуществляется с помощью текстового меню, для выбора варианта нужно ввести его номер.

Main menu:

“1) create new array” печатает меню создания массива. Необходимо указать положительный размер нового массива и выбрать способ инициализации, следуя выводимым указаниям.

“2) change array element” изменить значение элемента массива.

“3) sort array” печатает меню запуска сортировки. После завершения сортировки выводится число сравнений и перестановок сделанных программой.

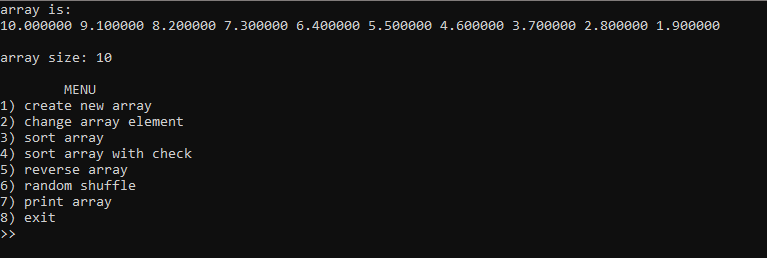
“4) sort array with check” печатает меню запуска сортировки. После завершения сортировки выводится число сравнений и перестановок сделанных программой и информация о корректности сортировки.

“5) reverse array” переставить элементы массива в обратном порядке.

“6) random shuffle” перемешать массив.

“7) print array” вывести весь массив (в консоль или в файл)

“8) exit” завершить работу программы.



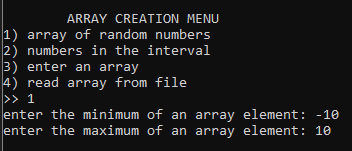
Array creation menu:

“1) array of random numbers” запрашивает два числа и создает массив случайных чисел в интервале между заданными значениями.

“2) numbers in the interval” запрашивает два значения полуинтервала и создает массив значений арифметической прогрессии.

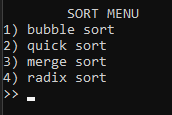
“3) enter an array” запрашивает ввод массива через консоль.

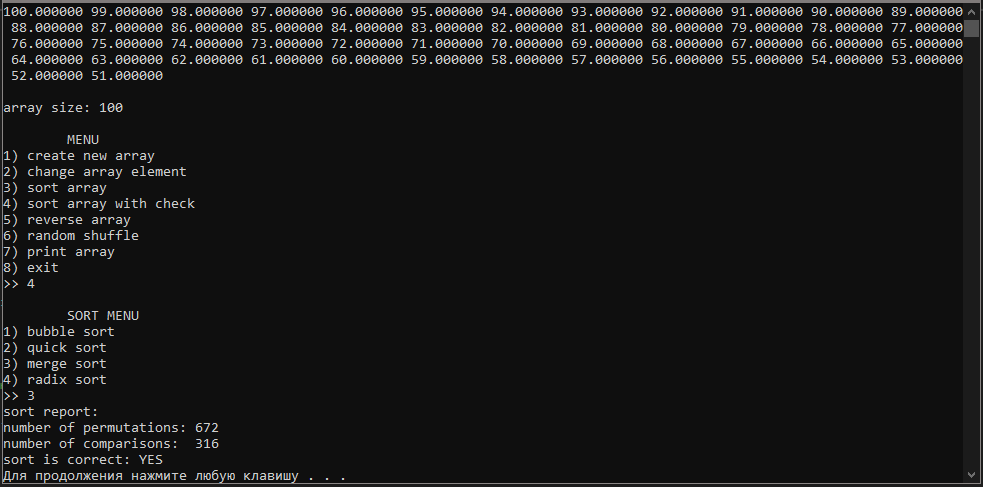
“4) read array from file” запрашивает имя текстового файла и считывает из него массив.



Sort menu

Запрашивает выбор алгоритма сортировки.





# Описание программной реализации

Файлы: sorts.h, sorts.c, menu.h, menu.c

Sorts.h прототипы функций сортировки и объявление структуры для подсчета числа перестановок и сравнений.

testInfo bubbleSort(float\* const arr, size\_t len);

testInfo mergeSort(float\* const arr, size\_t len);

testInfo quickSort(float\* const arr, size\_t len);

testInfo radixSort(float\* const arr, size\_t len);

testInfo checkSort(float\* const arr, size\_t len, testInfo(sortFunc)(float\* const, size\_t));

функции принимают в качестве аргументов указатель на начало массива, и количество элементов массива. Возвращается структура testInfo с полями compareCount равным числу сделанных сравнений элементов массива и swapCount равным числу перестановок элементов массива.

Sorts.c реализация функций сортировки.

Menu.h прототип функции mainMenu() - запуск консольного меню для взаимодействия с пользователем

menu.c реализация mainMenu(), дополнительные функции для работы mainMenu.

# Подтверждение корректности

Для подтверждения корректности в программе реализована функция checkSort, которая копирует массив и сравнивает результат работы сортировки с результатом работы функции qsort из стандартной библиотеки C.

testInfo checkSort(float\* const arr, size\_t len, testInfo(sortFunc)(float\* const, size\_t));

Аргументы: массив типа float, количество элементов массива, указатель на функцию сортировки.

Возвращаемое значение: структура testInfo с полями compareCount — число сравнений, swapCount — число перестановок, isCorrect — 1 если сортировка корректна, 0 — сортировка не корректна;

# Результаты экспериментов

Массив размера n случайные данные.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| len | перестановки пузырек | перестановки quickSort | перестановки mergeSort | перестановки radixSort |
| 2 | 1 | 1 | 2 | 9 |
| 4 | 1 | 2 | 8 | 19 |
| 8 | 17 | 6 | 24 | 37 |
| 16 | 58 | 16 | 64 | 75 |
| 32 | 260 | 41 | 160 | 153 |
| 64 | 1106 | 101 | 384 | 301 |
| 128 | 3892 | 211 | 896 | 605 |
| 256 | 16076 | 503 | 2048 | 1218 |
| 512 | 66154 | 1121 | 4608 | 2438 |
| 1024 | 260243 | 2457 | 10240 | 4868 |
| 2048 | 1032297 | 5340 | 22528 | 9737 |
| 4096 | 4196954 | 11832 | 49152 | 19432 |
| 8192 | 16930453 | 24807 | 106496 | 38933 |
| 16384 | 67297274 | 54359 | 229376 | 77853 |
| 32768 | 269308971 | 114894 | 491520 | 155685 |
| 65536 | 1073675542 | 240967 | 1048576 | 311251 |
| 131072 | 4287847082 | 504956 | 2228224 | 622628 |
| 262144 | 17241548711 | 1048979 | 4718592 | 1245231 |
| 524288 | 68760994274 | 2154861 | 9961472 | 2490422 |
| 1048576 | 274947975051 | 4281904 | 20971520 | 4980708 |
| 2097152 | 1099735883025 | 8658341 | 44040192 | 9961436 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| len | сравнения пузырек | сравнения quickSort | сравнения mergeSort | сравнения radixSort |
| 2 | 1 | 2 | 1 | 0 |
| 4 | 5 | 7 | 4 | 0 |
| 8 | 25 | 26 | 17 | 0 |
| 16 | 119 | 64 | 47 | 0 |
| 32 | 490 | 153 | 121 | 0 |
| 64 | 1845 | 376 | 296 | 0 |
| 128 | 7875 | 1252 | 736 | 0 |
| 256 | 32112 | 2304 | 1717 | 0 |
| 512 | 130771 | 5007 | 3960 | 0 |
| 1024 | 523755 | 12041 | 8968 | 0 |
| 2048 | 2092807 | 28535 | 19929 | 0 |
| 4096 | 8383074 | 55209 | 43946 | 0 |
| 8192 | 33542208 | 134173 | 96133 | 0 |
| 16384 | 134190426 | 278230 | 208633 | 0 |
| 32768 | 536845348 | 630561 | 449820 | 0 |
| 65536 | 2147391195 | 1383718 | 965719 | 0 |
| 131072 | 8589859186 | 2948728 | 2062588 | 0 |
| 262144 | 34359552350 | 6416640 | 4386390 | 0 |
| 524288 | 137438163450 | 14679068 | 9298232 | 0 |
| 1048576 | 549754901079 | 38243672 | 19645597 | 0 |
| 2097152 | 2199019383100 | 101715815 | 41389045 | 0 |

Массив размера len. Худший случай: опорным выбирается минимальный элемент подмассива, лучший случай: массив отсортирован.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| len | сравнения quickSort худший случай | перестановки quickSort худший случай | сравнения quickSort лучший случай | перестановки quickSort лучший случай |
| 2 | 2 | 1 | 2 | 1 |
| 4 | 9 | 3 | 6 | 2 |
| 8 | 35 | 7 | 17 | 4 |
| 16 | 135 | 15 | 46 | 8 |
| 32 | 527 | 31 | 119 | 16 |
| 64 | 2079 | 63 | 296 | 32 |
| 128 | 8255 | 127 | 713 | 64 |
| 256 | 32895 | 255 | 1674 | 128 |
| 512 | 131085 | 510 | 3853 | 257 |
| 1024 | 520945 | 1014 | 8734 | 521 |
| 2048 | 2061561 | 2012 | 19533 | 1056 |
| 4096 | 8172383 | 3995 | 43210 | 2142 |
| 8192 | 31668077 | 7722 | 94987 | 4475 |
| 16384 | 119871901 | 14614 | 207512 | 9475 |
| 32768 | 439953385 | 26830 | 451383 | 20300 |
| 65536 | 1535128777 | 46871 | 982945 | 44418 |
| 131072 | 5352647749 | 81656 | 2173569 | 101180 |
| 262144 | 19364740987 | 147660 | 4997351 | 229460 |
| 524288 | 73014199785 | 278509 | 12303107 | 491521 |
| 1048576 | 2,83359E+11 | 540563 | 33456389 | 1015808 |
| 2097152 | 1,11673E+12 | 1065045 | 101596008 | 2064384 |

Массив размера len. Лучший случай: массив отсортирован, худший случай: массив отсортирован в обратном порядке.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| len | сравнения пузырек худший случай | перестановки пузырек худший случай | сравнения пузырек лучший случай | перестановки пузырек лучший случай |
| 2 | 1 | 1 | 1 | 0 |
| 4 | 6 | 6 | 3 | 0 |
| 8 | 28 | 28 | 7 | 0 |
| 16 | 120 | 120 | 15 | 0 |
| 32 | 496 | 496 | 31 | 0 |
| 64 | 2016 | 2016 | 63 | 0 |
| 128 | 8128 | 8128 | 127 | 0 |
| 256 | 32640 | 32639 | 255 | 0 |
| 512 | 130816 | 130814 | 511 | 0 |
| 1024 | 523776 | 523758 | 1023 | 0 |
| 2048 | 2096128 | 2096070 | 2047 | 0 |
| 4096 | 8386560 | 8386358 | 4095 | 0 |
| 8192 | 33550336 | 33549329 | 8191 | 0 |
| 16384 | 134209536 | 134205470 | 16383 | 0 |
| 32768 | 536854528 | 536838190 | 32767 | 0 |
| 65536 | 2147450880 | 2147385297 | 65535 | 0 |
| 131072 | 8589869046 | 8589607426 | 131071 | 0 |
| 262144 | 34359607275 | 34358558856 | 262143 | 0 |
| 524288 | 1,37439E+11 | 1,37434E+11 | 524287 | 0 |
| 1048576 | 5,49755E+11 | 5,49739E+11 | 1048575 | 0 |
| 2097152 | 2,19902E+12 | 2,19896E+12 | 2097151 | 0 |

По данным экспериментов видно, что сортировки имеют следующую сложность:

* сортировка пузырьком: O(n) в лучшем случае, O(n^2) в худшем и среднем.
* быстрая сортировка: O(n \* ln(n)) в лучшем и среднем, O(n^2) в худшем случае.
* Сортировка слиянием: O(n \* ln(n)) сложность зависит, только от размера массива.
* Поразрядная сортировка: O(n) сложность зависит, только от размера массива.

# Заключение

Были реализованы предложенные функции сортировки и функция проверки корректности сортировок. Получено экспериментальное подтверждение теоретической сложности алгоритмов.

# Приложение

#pragma comment(linker, "/STACK:256000000")

#include "sorts.h"

#include "stdlib.h"

#include "inttypes.h"

#include "memory.h"

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

{

float temp = \*a;

\*a = \*b;

\*b = temp;

}

static void ptrSwap(float\*\* a, float\*\* b)

{

float\* temp = \*a;

\*a = \*b;

\*b = temp;

}

testInfo bubbleSort(float\* const arr, size\_t len)

{

int flag = 1;

testInfo report;

report.swapCount = report.compareCount = 0;

for (size\_t i = 0; i < len && flag; i++)

{

flag = 0;

for (size\_t j = len - 1; j > i; j--)

{

report.compareCount++;

if (arr[j] < arr[j - 1]) {

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

report.swapCount++;

flag = 1;

}

}

}

return report;

}

static int64\_t \_partition(float\* arr, int64\_t left, int64\_t right, testInfo\* report)

{

// pos - псевдослучайное 32-х битное число на [left, right)

#if RAND\_MAX <= (1 << 16)

size\_t pos = (rand() | (rand() << 15)) % (right - left) + left;

#else

size\_t pos = rand() % (right - left) + left;

#endif

float pivot = arr[pos];

while (left < right)

{

while (left <= right && arr[left] <= pivot)

{

report->compareCount++;

left++;

}

while (left <= right && arr[right] >= pivot)

{

report->compareCount++;

right--;

}

if (left < right) {

report->swapCount++;

swap(&arr[left], &arr[right]);

}

}

report->swapCount++;

if (pos > left) {

swap(&arr[pos], &arr[left]);

return left;

}

swap(&arr[pos], &arr[left - 1]);

return left - 1;

}

static void \_quickSort(float\* arr, int64\_t first, int64\_t last, testInfo\* report)

{

if (first < last) {

int64\_t p = \_partition(arr, first, last, report);

\_quickSort(arr, first, p - 1, report);

\_quickSort(arr, p + 1, last, report);

}

}

testInfo quickSort(float\* const arr, size\_t len)

{

testInfo report;

report.compareCount = report.swapCount = 0;

\_quickSort(arr, 0, (int64\_t)len - 1, &report);

return report;

}

static void \_merge(float\* arr, float\* buffer, size\_t l, size\_t r, size\_t mid, testInfo\* report)

{

size\_t i = l, j = mid + 1, k = l;

while (i <= mid || j <= r)

{

if (i <= mid && j <= r) {

report->compareCount++;

}

if (j > r || (i <= mid && arr[i] < arr[j])) {

buffer[k] = arr[i];

i++;

}

else {

buffer[k] = arr[j];

j++;

}

k++;

}

}

static void \_mergeSort(float\* arr, float\* buffer, size\_t l, size\_t r, testInfo\* report)

{

if (l < r) {

size\_t mid = l + (r - l) / 2;

// при рекурсивном вызове arr и buffer меняются местами

\_mergeSort(buffer, arr, l, mid, report);

\_mergeSort(buffer, arr, mid + 1, r, report);

\_merge(buffer, arr, l, r, mid, report);

}

}

testInfo mergeSort(float\* const arr, size\_t len)

{

testInfo report;

float\* buffer = (float\*)malloc(len \* sizeof(float));

report.compareCount = report.swapCount = 0;

for (int i = 0; i < len; i++)

{

buffer[i] = arr[i];

}

\_mergeSort(arr, buffer, 0, len - 1, &report);

free(buffer);

return report;

}

static void \_radixNormalizeInPlace(float\* arr, size\_t len, testInfo\* report)

{

if (arr[len - 1] > 0) return;

report->swapCount += len / 2;

for (size\_t i = 0; i < len/2; i++)

{

swap(&arr[i], &arr[len - i - 1]);

}

size\_t j = 0;

while (j < len && arr[j] < 0) j++;

report->swapCount += (len + j) / 2 - j;

for (size\_t i = j; i < (len + j) / 2; i++)

{

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

}

}

testInfo radixSort(float\* const arr, size\_t len)

{

const int countValues = (1 << 8);

testInfo report;

size\_t counts[sizeof(float)][countValues];

float\* tempArr;

float\* buffer;

memset(counts, 0, sizeof(float) \* countValues \* sizeof(size\_t));

report.compareCount = report.swapCount = 0;

for (int byteNumber = 0; byteNumber < sizeof(float); byteNumber++)

{

for (size\_t i = 0; i < len; i++)

{

int val = ((uint8\_t\*)&arr[i])[byteNumber];

counts[byteNumber][val]++;

}

}

for (int byteNumber = 0; byteNumber < sizeof(float); byteNumber++)

{

size\_t sum = 0;

for (int i = 0; i < countValues; i++)

{

sum += counts[byteNumber][i];

counts[byteNumber][i] = sum - counts[byteNumber][i];

}

}

tempArr = arr;

buffer = (float\*)malloc(sizeof(float) \* len);

for (int byteNumber = 0; byteNumber < sizeof(float); byteNumber++)

{

for (size\_t i = 0; i < len; i++)

{

int val = ((uint8\_t\*)&tempArr[i])[byteNumber];

buffer[counts[byteNumber][val]] = tempArr[i];

counts[byteNumber][val]++;

report.swapCount++;

}

ptrSwap(&buffer, &tempArr);

}

if (arr == tempArr) {

\_radixNormalizeInPlace(arr, len, &report);

free(buffer);

}

else {

\_radixNormalize(arr, tempArr, len);

report.swapCount += len;

free(tempArr);

}

return report;

}

testInfo checkSort(float\* const arr, size\_t len, testInfo(sortFunc)(float\* const, size\_t))

{

float\* arrCopy = (float\*)malloc(sizeof(float) \* len);

for (int i = 0; i < len; i++)

{

arrCopy[i] = arr[i];

}

testInfo report = sortFunc(arr, len);

qsort(arrCopy, len, sizeof(float), compare);

report.isCorrect = 1;

for (int i = 0; i < len; i++)

{

if (arr[i] != arrCopy[i]) {

report.isCorrect = 0;

break;

}

}

free(arrCopy);

return report;

}