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**

**«Алгоритмы сортировки массивов»**

**Выполнил**:

студентка группы 382003-1

Капустина П.Д.

**Проверил**:

ассистент каф. МОСТ,

Волокитин В.Д.

Нижний Новгород

2019

**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 сравнить четыре различных алгоритма сортировки (сортировка пузырьком, сортировка расческой, сортировка слиянием, поразрядная сортировка) для вещественных чисел типа double. Замерить число перестановок и сравнений, выполненных при исполнении программы. Экспериментально подтвердить теоретическую сложность алгоритмов

# Метод решения

*Сортировка пузырьком:*

Алгоритм состоит в повторяющихся проходах по сортируемому массиву. На каждой итерации последовательно сравниваются соседние элементы, и, если порядок в паре неверный, то элементы меняют местами. Также существует переменная flag, которая в случае отсортировки массива позволяет покинуть функцию. Проход по массиву происходит при помощи двух циклов, второй является вложенный в первый. Первый цикл пробегает по значениям от 0 до N, которое является числом элементов в массиве. Второй цикл пробегает по значениям от N-1 до Т, где Т – значение шага первого цикла. Перед началом второго цикла значению flag присваивается 0, в случае перестановок во втором цикле flag становится равен 1. Во втором цикле два соседних элемента сравниваются между собой и меняются местами, если левый оказывается больше правого. Если после второго цикла flag = 0, то массив отсортирован и функция прекращает работу.

*Сортировка расческой:*

Производятся неоднократные прогоны по массиву, при которых сравниваются пары элементов. Если они неотсортированные друг относительно друга - то производится обмен. В результате крупные элементы мигрируют в конец массива, а небольшие по значению - в начало. Сравниваются элементы, между которыми некоторое фиксированное расстояние. При каждом следующем прохождении это расстояние уменьшается, пока не достигнет минимальной величины. Уменьшающееся расстояние между сравниваемыми элементами рассчитывается с помощью специальной величины, называемой фактором уменьшения. Длина массива делится на этот фактор, это и есть разрыв между индексами. После каждого прохода расстояние делится на фактор уменьшения и таким образом получается новое значение. В конце концов оно сужается до минимального значения - единицы, и массив просто отсортировывается обычным "пузырьком".

*Сортировка слиянием:*

Рекурсивное разбиение задачи на меньшие происходит до тех пор, пока размер массива не достигнет единицы (любой массив длины 1 можно считать упорядоченным). Соединение двух упорядоченных массивов в один и их слияние в третий результирующий массив. На каждом шаге мы берём меньший из двух первых элементов подмассивов и записываем его в результирующий массив. Счётчики номеров элементов результирующего массива и подмассива, из которого был взят элемент, увеличиваем на 1. Когда один из подмассивов закончился, мы добавляем все оставшиеся элементы второго подмассива в результирующий массив. Эти действия повторяются до тех пор, пока длина результирующего массива не становится равной длине первоначального массива.

*Поразрядная сортировка:*

Элементы массива сортируются по значению одного байта из их преставления в памяти компьютера, байт выбирается последовательно от младшего к старшему. При каждом проходе осуществляется сортировка подсчетом для каждой цифры справа налево. Подсчитывается количество элемента массива со всеми возможными значениями байта. Нахождение чисел с новым значением байта является количеством элементов предыдущими значениями байтов. Числа изначального массива записываются на полученные позиции. После записи та увеличивается на единицу.

# Руководство пользователя

Пользователь должен ввести одно число от 0 до 8, в зависимости от того, что ему нужно.

При вводе цифры 1 программа предлагает пользователю ввести массив вручную, число элементов в этом массиве указано в конце третьего пункта. Пользователь должен ввести n чисел через пробел или нажимая «Enter» после каждого ввода.

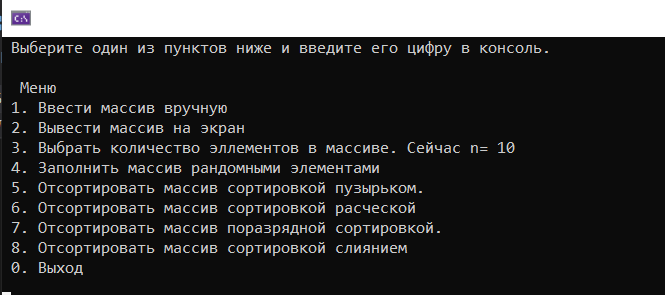
При вводе цифры 2 программа выводит массив на экран консоли.

При вводе цифры 3 пользователь должен ввести одно число, которое потом будет являться количеством элементов в массиве.

При вводе цифры 4 массив заполняется различными элементами вещественного типа.

При вводе цифр 5-8 массив отсортировывается и на экран выводится количество сравнений и перестановок, которые были совершены сортировкой.

При вводе цифры 0 программа перестает работать.



.

# Описание программной реализации

В программе содержатся следующие функции и структуры:

**testInfo –** структура, содержащая в себе два значения типа int64\_t, создана для того, чтобы было удобнее возвращать количество перестановок и сортировок из функций.

**bubble\_sort** – сортировка пузырьком, проходит по массиву в двух циклах и отсортировывает его элементы по возрастанию, сравнивая соседние между собой. В качестве аргументов принимает указатель на массив типа double и его размер типа const. Возвращает структуру *testInfo,* в которой хранится число перестановок и сравнений.

**comb\_sort –** сортировка расческой. Производятся неоднократные прогоны по массиву, при которых сравниваются пары элементов. Если они неотсортированные друг относительно друга - то производится обмен. В конце происходит один прогон «пузырьком». В качестве аргументов принимает указатель на массив типа double и его размер типа const. Возвращает структуру *testInfo,* в которой хранится число перестановок и сравнений сортировки.

**mergeSort –** сортировка слиянием. Использует рекурсию, разбивая массив на более маленькие массивы, вплоть до единичного, а потом сливает их в один сортируя элементы в порядке возрастания. В качестве аргументов принимает указатель на массив типа double, номер первого элемента и размер массива типа int. Возвращает структуру *testInfo,* в которой хранится число перестановок и сравнений сортировки.

**radixSort –** поразрядная сортировка. Элементы массива сортируются по значению одного байта из их преставления в памяти компьютера, байт выбирается последовательно от младшего к старшему. В качестве аргументов принимает указатель на массив типа double и его размер типа size\_t. Возвращает структуру *testInfo,* в которой хранится число перестановок и сравнений сортировки.

**massiv –** заполняет передаваемый массив рандомными значениями. В качестве аргументов принимает указатель на массив типа double и его размер типа size\_t. Ничего не возвращает.

**сorrect –** проверяет сортировку на корректность исполнения. В качестве аргументов принимает указатель на массив типа double и его размер типа size\_t. Ничего не возвращает.

# Подтверждение корректности

Для подтверждения корректности в программе присутствует проверка с помощью функции **сorrect.** Она получает указатель на массив и размер массива, а после сравнивает соседние элементы между собой. Если значение элемента массива под номером i оказывается меньше значения элемента массива под номером i+1, то значение переменной flag приравнивается 0, иначе оно равно 1. Дальше функция вызывается в программе и в зависимости от значения flag выдает результат. Корректно, если flag=1, и не корректно, если flag=0.

# Результаты экспериментов

*Сортировка пузырьком:*

Теоретическая сложность алгоритма – O(n^2). При сортировке массивов разных размеров лучшим для сортировки будет тот, в котором входной массив уже отсортирован, худший – массив отсортирован в обратном порядке. Число сравнений в худшем случае всегда приближается к теоретическому N\*(N-1)/2, поэтому можно считать, что сортировка была реализована верно.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 10 | 50 | 100 | 500 | 1000 | 5000 | 10000 |
| Лучший | 9 | 476 | 2243 | 59349 | 232038 | 6163483 | 24838718 |
| Средний | 21 | 601 | 2503 | 62356 | 249046 | 6264209 | 25095477 |
| Худший | 38 | 777 | 2730 | 66409 | 261358 | 6379655 | 25208607 |

Перестановки:

Сравнения:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 10 | 50 | 100 | 500 | 1000 | 5000 | 10000 |
| Лучший | 30 | 1120 | 4778 | 123372 | 494940 | 12478194 | 49976279 |
| Средний | 41 | 1192 | 4890 | 124359 | 498608 | 12493611 | 49988715 |
| Худший | 45 | 1224 | 4994 | 124744 | 499479 | 12497175 | 49994790 |

*Сортировка расческой:*

Теоретическая сложность алгоритма зависит от рассматриваемого случая. При сортировке массивов разных размеров лучшим для сортировки будет тот, в котором входной массив уже отсортирован и его сложность O(n\*log n), в среднем – O(n^2/2^p), где p – количество приращений, в худшем - O(n^2). Благодаря эксперименту мы видим, что сложность алгоритма совпадает с теоретической.

Перестановки:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 10 | 50 | 100 | 500 | 1000 | 5000 | 10000 |
| Лучший | 12 | 413 | 2050 | 56716 | 239629 | 6128338 | 24694242 |
| Средний | 23 | 612 | 2454 | 62170 | 249139 | 6250461 | 25062627 |
| Худший | 32 | 773 | 2887 | 66772 | 260539 | 6418995 | 25384173 |

Сравнения:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 10 | 50 | 100 | 500 | 1000 | 5000 | 10000 |
| Лучший | 48 | 1705 | 8064 | 214126 | 893162 | 24160061 | 96909994 |
| Средний | 93 | 2418 | 9715 | 241787 | 972588 | 24667876 | 98949058 |
| Худший | 120 | 2881 | 11034 | 257539 | 1016039 | 25034886 | 100239661 |

*Сортировка слиянием:*

Теоретическая сложность алгоритма – O(n\*log n). Время работы этого алгоритма не зависит от того отсортирован массив или нет, поэтому чем больше числа, тем меньше разницы между «худшим» и «лучшим» случаем. Благодаря эксперименту мы видим, что сложность алгоритма совпадает с теоретической.

Перестановки:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 10 | 50 | 100 | 500 | 1000 |
| Лучший | 8 | 117 | 286 | 2111 | 4738 |
| Средний | 15 | 135 | 319 | 2156 | 4832 |
| Худший | 24 | 156 | 358 | 2195 | 4900 |

Сравнения:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 10 | 50 | 100 | 500 | 1000 |
| Средний | 34 | 286 | 672 | 4488 | 9976 |

*Сортировка поразрядная (O(n\*log n)):*

Сравнения в этой сортировке не используются, а количество перестановок не зависит от входных данных. Через эксперимент мы смогли это подтвердить.

Перестановки:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 10 | 50 | 100 | 500 | 1000 | 5000 | 10000 |
| Все | 80 | 400 | 800 | 4000 | 8000 | 40000 | 80000 |

# Заключение

Были реализованы четыре различные сортировки, их работа была проверена на корректность, и та подтвердилась. Экспериментально было проверено число перестановок и сравнений алгоритмов. Экспериментальные данные совпали или были достаточно близки к теоретическим.

# Приложение

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <time.h>

#include <conio.h>

#include <malloc.h>

#include <stdlib.h>

#include <math.h>

#include "inttypes.h"

#include "memory.h"

#include <locale.h>

#define SIZE\_T 100000

struct testInfo

{

int64\_t compareCount, swapCount;

int isCorrect;

};

typedef struct testInfo testInfo;

void createCounters(double\* array, long\* counters, int N)

{

int i;

unsigned char\* bp = (unsigned char\*)array;

unsigned char\* Dataend = (unsigned char\*)(array + N);

memset(counters, 0, 256 \* sizeof(double) \* sizeof(int));

while (bp != Dataend)

{

for (i = 0; i < sizeof(double); i++)

{

counters[256 \* i + \*(bp++)]++;

}

}

}

testInfo floatradixLastPass(short Offset, int size, double\* array, double\* dest, long\* count)

{

testInfo radix;

radix.swapCount = radix.compareCount = 0;

double\* sp;

long s, c, i, \* cp;

unsigned char\* bp;

long numNeg = 0;

for (i = 128; i < 256; i++) numNeg += count[i];

s = numNeg;

cp = count;

for (i = 0; i < 128; ++i, ++cp)

{

c = \*cp;

\*cp = s;

s += c;

}

cp = count + 255;

s = 0;

\*cp = 0;

for (i = 254; i >= 128; i--, cp--)

{

s += \*cp;

\*cp = s;

}

bp = (unsigned char\*)array + Offset;

sp = array;

for (i = size; i > 0; --i, bp += sizeof(double), ++sp)

{

cp = count + \*bp;

if (\*bp < 128)

{

dest[\*cp] = \*sp;

(\*cp)++;

}

else

{

(\*cp)--;

dest[\*cp] = \*sp;

}

radix.swapCount++;

}

return radix;

}

testInfo radixPass(short Offset, int N, double\* sourse, double\* dest, long\* count)

{

testInfo radix;

radix.swapCount = radix.compareCount = 0;

double\* sp;

long s, c, i, \* cp;

unsigned char\* bp;

s = 0;

cp = count;

for (i = 256; i > 0; --i, ++cp)

{

c = \*cp;

\*cp = s;

s += c;

}

bp = (unsigned char\*)sourse + Offset;

sp = sourse;

for (i = N; i > 0; --i, bp += sizeof(double), ++sp)

{

cp = count + \*bp;

dest[\*cp] = \*sp;

(\*cp)++;

radix.swapCount++;

}

return radix;

}

testInfo FloatRadixSort(double\* array, int size)

{

testInfo radix;

testInfo rad;

radix.swapCount = radix.compareCount = 0;

int i;

double\* temp;

double\* mas = (double\*)malloc(sizeof(double) \* size);

long\* counters = (long\*)malloc(sizeof(double) \* 256 \* sizeof(int));

long\* count;

createCounters((double\*)array, counters, size);

for (i = 0; i < sizeof(double) - 1; i++)

{

count = counters + 256 \* i;

rad = radixPass(i, size, array, mas, count);

radix.swapCount += rad.swapCount;

radix.compareCount += rad.compareCount;

temp = array;

array = mas;

mas = temp;

}

count = counters + 256 \* (sizeof(double) - 1);

rad = floatradixLastPass(sizeof(double) - 1, size, array, mas, count);

radix.swapCount += rad.swapCount;

radix.compareCount += rad.compareCount;

temp = array;

array = mas;

mas = temp;

free(mas);

free(counters);

return radix;

}

double massiv(double\* array, int l)

{

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

{

array[i] = rand();

array[i] = (array[i] \* 0.00001 + rand());

if ((rand() % 2) == 1)

array[i] = 0 - array[i];

}

}

testInfo bubble\_sort(double\* a, const size)

{

testInfo bubble;

bubble.compareCount = bubble.swapCount = 0;

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

{

int flag = 0;

for (int j = size - 2; j > i - 1; j--)

{

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

{

double tmp;

tmp = a[j];

a[j] = a[j + 1];

a[j + 1] = tmp;

bubble.swapCount++;

flag = 1;

}

bubble.compareCount++;

}

if (flag == 0)

{

return bubble;

break;

}

}

}

testInfo comb\_sort(double\* a, const size)

{

testInfo comb;

comb.compareCount = comb.swapCount = 0;

double factor = 1.2473309;

int step = size - 1;

while (step >= 1) {

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

{

if (a[i] > a[i + 1])

{

double tmp;

tmp = a[i];

a[i] = a[i + 1];

a[i + 1] = tmp;

comb.swapCount++;

}

comb.compareCount++;

}

step /= factor;

}

int b = 1;

while (b == 1) {

b = 0;

for (int j = 0; j + 1 < size; j++)

{

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

{

double tmp;

tmp = a[j];

a[j] = a[j + 1];

a[j + 1] = tmp;

b = 1;

comb.swapCount++;

}

comb.compareCount++;

}

}

return comb;

}

testInfo mergeSort(double\* a, int l, int r)

{

testInfo merge;

merge.compareCount = merge.swapCount = 0;

if (l == r) return merge;

int mid = (l + r) / 2;

merge.swapCount += mergeSort(a, l, mid).swapCount;

merge.swapCount += mergeSort(a, mid + 1, r).swapCount;

merge.compareCount += mergeSort(a, l, mid).compareCount;

merge.compareCount += mergeSort(a, mid + 1, r).compareCount;

int i = l;

int j = mid + 1;

float\* tmp = (float\*)malloc(r \* sizeof(int));

for (int step = 0; step < r - l + 1; step++)

{

if ((a[i] > a[j]))

merge.swapCount++;

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

{

tmp[step] = a[i];

i++;

}

else

{

tmp[step] = a[j];

j++;

}

merge.compareCount++;

}

for (int step = 0; step < r - l + 1; step++)

{

a[l + step] = tmp[step];

}

for (int step = 0; step < r - l + 1; step++)

return merge;

}

int correct(double\* array, int l)

{

int flag = 1;

for (int i = 0; i < l-1; i++)

{

if (array[i] > array[i + 1])

flag = 0;

}

return flag;

};