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

Митин Р. А.

**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

Необходимо было реализовать несколько алгоритмов сортировки: сортировка слиянием, сортировка хоара, поразрядная сортировка и сортировка выбором для типа данных float на языкe C. Провести их анализ, с помощью подсчёта количества перестановок и сравнений.

# Метод решения

Сортировка выбором:

Сортировка выбором делит массив на 2 части: отсортированную и не отсортированную. Изначально сортированная часть пустая. Далее на каждой итерации внешнего цикла к сортированной части добавляется один элемент, происходит это так:

Алгоритм, с помощью среднего цикла, находит в неотсортированной части минимальный элемент и затем меняет его местами с первым элементом в неотсортированной части. Таким образом за полный проход внешнего цикла все элементы встают на свои места.

Сортировка хоара (быстрая сортировка):

Сортировка хоара состоит из 2 функций: первая – partition, она берёт в качестве ведущего элемента правый элемент из части, в которой она работает, и, используя цикл while, идёт по массиву с права и слева, если слева находиться элемент больше или равный ведущему, то он меняется местами с элементом из правой части, который меньше ведущего. Таким образом после завершения работы цикла while массив разделён на 2 части: в одной части все элементы меньше или равны ведущему, а в другой все больше ведущего. После завершения цикла while функция меняет местами ведущий элемент и крайний левый элемент в правой части. Далее функция возвращает позицию ведущего элемента. Вторая функция – sortxoar, в первую очередь она проверяет размер части, с которой работает, если часть размером один, то функция ничего не делает, если нет, то она вызывает функцию partition после чего вызывает себя же, но уже для левой (все элементы меньше или равны ведущему и правой (все элементы больше ведущего) части, таким образом полностью рекурсивно упорядочивая массив.

Сортировка слиянием:

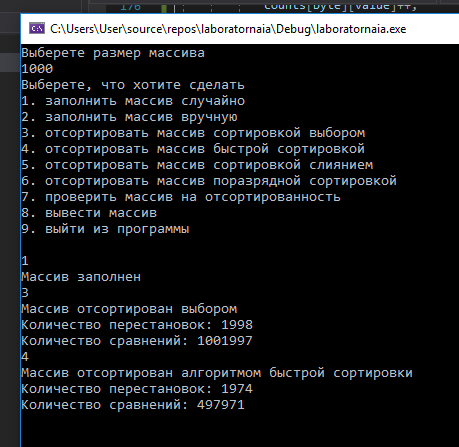
Сортировка слиянием сначала проверяет размер части, с которой она работает, если размер равен 1, то часть отсортирована и ничего делать не надо, если нет, то функция разбивает выбранную часть на 2 примерно равные и вызывает себя рекурсивно уже для двух новых частей. Далее с помощью цикла while и двух циклов for, два отсортированных массива сливаются в один отсортированный, находящийся в дополнительной памяти. И последним циклом новый отсортированный массив перемещается из дополнительной памяти в основную.

Поразрядная сортировка:

Сначала алгоритм создаёт дополнительный двумерный массив размером [sizeof(float)][256] и заполняет его нулями, он необходим в дальнейшем для устойчивой сортировки посчётом. Далее алгоритм с помощью двух циклов for проходит по массиву считая количество значений конкретных байт у всех чисел. По итогу двух циклов в массиве counts[i][j] будет количество чисел, у которых в i-том байте значение равно j. Следующие два цикла for меняют массив counts слудующим образом: в counts[i][j] будет записан индекс числа с i-ым байтом равном j. Эти два цикла были необходимы, что бы сортировка подсчётом была устойчивой. После этого выделяется дополнительная память размером sizeof(float)\*N. Два следующих цикла for побайтого заполняют изначальный массив уже в отсортированном порядке, не считая отрицательных чисел. Во всех предыдущих случаях внутренний цикл for работал с одним конкретным байтом у каждого числа, так как байтов несколько, то для полной сортировки необходимо было пройти по каждому байту. Так как в битовом представлении у всех отрицательных чисел старший бит 1, то и старший байт у всех отрицательных чисел больше, следовательно, после данной сортировки все отрицательные числа будут находить в конце массива, причём уже в правильном порядке. Что бы завершить сортировку, алгоритм с помощью цикла while считает количество положительных чисел в массиве, после чего используя 2 цикла for переворачивает сначала все положительные числа, а затем и весь массив. После этих операций исходный массив будет полностью отсортирован.

# Руководство пользователя

При запуске программы пользователь должен ввести размер массива, с которым он хочет работать. Далее перед пользователем появляется меню, в котором написано какое число, что обозначает. При вводе числа не описанного в меню программа завершит работу. При вводе двойки необходимо отдельно ввести каждый элемент из массива. При вводе любого другого числа программа выведет какое-либо сообщение, подтверждающее выполнение данной команды. Что бы изменить размер массива необходимо перезапустить программу.



# Описание программной реализации

Всего в проекте содержится 5 файлов, 2 из них заголовочные и имеют расширение .h, другие 3 содержат реализацию необходимых для работы функций.

Файл main.c содержит две функции draw() – выводит на экран меню, и scan(float\* a,int N) – считывает число, введённое пользователем и вызывает соответствующие функции из других файлов. Сам main.c лишь создаёт массив, делает возможным писать в консоли на русском и в цикле вызывает функцию scan.

Файл rabotasmassivami.c включает в себя несколько функций для работы с массивами sozdaniemassiva(float\*, int) – заполняет массив случайными числами, vivodmassiva(float\*, int) и vivodmassiva2(float\*, int) – используются для вывода массива, первая функция выводить массив в строчку(использовать при небольших размерах), вторая в столбец; функция zapolnenievrucnuu(float\*, int) – реализует возможность ручного заполнения массива. int proverkamassiva(float\* a, float\* c, int N) – проверяет отсортирован ли массив а, если да возвращает значение 1, если нет 0. arraycopy(float\*, float\*, int) – копирует значения из первого массива во второй. Все функции (кроме proverkamassiva) из этого файла имеют тип возвращаемого значения void.

Файл sortirovki.c реализует сортировки, используемые в программе, их подробное описание можно прочитать в разделе Метод решения, здесь будут представлены только прототипы функций: int partition (float\*, int, int) sortxoar(float\*, int, int) mergesort(float\*, float\*, int, int) sortvibor(float\*, int) void radixsort(float \*a, int N). Единственная неописанная функция sortbubble(float\* a, int N) – реализует сортировку пузырьком для проверки массива на отсортированность. Все функции(кроме partition) из этого файла имеют тип возвращаемого значения void.

# Подтверждение корректности

Для подтверждения корректности в программе используется функция proverkamassiva, при вводе новых данных в основной массив, создаётся побочный массив, в котором хранятся исходные данные, при вызове функции proverkamassiva – в него предаются оба массива, побочный массив сортируется пузырьковой сортировкой и поэлементно сравнивается с основным массивом, если массивы совпадают, все функции сортировки отработали корректно.

# Результаты экспериментов

В таблице представлены результаты экспериментов. Проведены четыре вида сортировки на разных размерах массива, замерено количество перестановок и сравнений. Так же в таблице посчитаны некоторые значения необходимые для подтверждения оценки асимптотической сложности алгоритмов. Для подсчёта были использованы следующие формулы:

Для сортировки выбором:

, где P1 и S1 – количество перестановок и сравнений в 1 эксперименте

Nk и N1 – количество элементов в массиве в 1 и к эксперименте

Pk и Sk – количество перестановок и сравнений в k эксперименте

Теоретически при F должно стремиться к константе. Так и происходит при больших N значение функции всё меньше меняется, что подтверждает верность асимптотической оценки.

Для сортировки хоара:

, где P1 и S1 – количество перестановок и сравнений в 1 эксперименте

Nk и N1 – количество элементов в массиве в 1 и к эксперименте

Pk и Sk – количество перестановок и сравнений в k эксперименте

Теоретически при F должно стремиться к константе. Однако на практике это не совсем так. Это вызвано тем, что при выборе ведущего элемента не всегда попадается лучший вариант, от того насколько удачно выбран ведущий элемент зависит основание логарифма, что немного влияет на значение функции, тем не менее они довольно близки друг к другу, что бы сделать вывод о верности асимптотической оценки.

Для сортировки слиянием:

, где P1 и S1 – количество перестановок и сравнений в 1 эксперименте

Nk и N1 – количество элементов в массиве в 1 и к эксперименте

Pk и Sk – количество перестановок и сравнений в k эксперименте

Теоретически при F должно стремиться к константе. Однако на практике это не совсем так. Как и в предыдущем алгоритме это вызвано тем, что для разных входных данных алгоитм работает с разной эффективностью, это влияет на основание логарифма, тем не менее они довольно близки друг к другу, что бы сделать вывод о верности асимптотической оценки.

Для поразрядной сортировки:

, где P1 и S1 – количество перестановок и сравнений в 1 эксперименте

Nk и N1 – количество элементов в массиве в 1 и к эксперименте

Pk и Sk – количество перестановок и сравнений в k эксперименте

Теоретически при F должно стремиться к константе, и из данных видно, что при больших N значения значение функции всё меньше меняется, что подтверждает верность асимптотической оценки.

По данным экспериментов видно, что асимптотическую сложность алгоритмов можно использовать в практических целях и при достаточно больших N оценивать время работы алгоритмов.

# Заключение

Реализовано несколько алгоритмов сортировки: сортировка слиянием, сортировка хоара, поразрядная сортировка и сортировка выбором для типа данных float на языкe C. Проведён их анализ, с помощью подсчёта количества перестановок и сравнений. Результаты экспериментов подтверждают верность теоретической оценки алгоритмов, это означает, что асимптотическую сложность алгоритмов можно использовать в практических целях и при больших N оценивать время работы алгоритмов.

# Приложение

int partition(float\* a, int l, int r)

{

float o = a[r];

int position = r;

while (l < r)

{

while (a[l] < o)

{

l++;

sravnenia++;

}

while (a[r] >= o)

{

r--;

sravnenia++;

}

sravnenia+=2;

if (l >= r)

break;

float tmb = a[l];

a[l] = a[r];

a[r] = tmb;

perestanovki += 2;

}

float t = a[l];

a[l] = o;

a[position] = t;

perestanovki+=2;

return l;

}

void sortxoar(float\* a, int l, int r)

{

if (l < r)

{

sravnenia++;

int p = partition(a, l, r);

sortxoar(a, l, p - 1);

sortxoar(a, p + 1, r);

}

}

void mergesort(float\* a, float\* b, int l, int r)

{

if (l < r)

{

int mid = (l + r) / 2;

mergesort(a, b, l, mid);

mergesort(a, b, mid + 1, r);

int leftindex = l;

int rightindex = mid + 1;

int i = l;

while (leftindex <= mid && rightindex <= r)

{

if (a[leftindex] <= a[rightindex])

{

b[i] = a[leftindex];

leftindex++;

}

else

{

b[i] = a[rightindex];

rightindex++;

}

i++;

sravnenia+=2;

perestanovki++;

}

for (int j = leftindex; j <= mid; j++)

{

b[i] = a[j];

i++;

perestanovki++;

sravnenia++;

}

for (int j = rightindex; j <= r; j++)

{

sravnenia++;

b[i] = a[j];

i++;

perestanovki++;

}

for (int j = l; j <= r; j++)

{

sravnenia++;

a[j] = b[j];

perestanovki++;

}

}

}

void sortvibor(float\* a,int N)

{

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

{

float min = a[i];

int position = i;

for (int j = i; j < N; j++)

{

if (a[j] < min)

{

min = a[j];

position = j;

}

sravnenia+=2;

}

float t = a[i];

a[i] = min;

a[position] = t;

perestanovki+=2;

sravnenia++;

}

}

void radixsort(float\* a, int N)

{

int sizeofbyte = 256;

int counts[sizeof(float)][256];

float\* temp;

float\* b;

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

{

for (int j = 0; j < 256; j++)

{

counts[i][j] = 0;

}

}

sravnenia = 256 \* sizeof(float);

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

{

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

{

int value = ((unsigned char\*)(a+i))[byte];

counts[byte][value]++;

sravnenia++;

}

sravnenia++;

}

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

{

int sum = 0;

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

{

sum += counts[byte][i];

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

sravnenia++;

}

sravnenia++;

}

temp = a;

if (NULL == (b = (float\*)malloc(sizeof(float) \* N)))

{

exit(-1);

}

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

{

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

{

int value = ((unsigned char\*)(temp+i))[byte];

b[counts[byte][value]] = temp[i];

counts[byte][value]++;

perestanovki++;

sravnenia++;

}

sravnenia++;

float\* t = b;

b = temp;

temp = t;

}

int amountofpositive = 0;

while (a[amountofpositive] > 0 && amountofpositive < N)

{

amountofpositive++;

sravnenia++;

}

int i, j;

for (i = 0; i < (amountofpositive / 2); i++)

{

j = amountofpositive - i - 1;

float t = a[i];

a[i] = a[j];

a[j] = t;

perestanovki += 2;

sravnenia++;

}

i = j = 0;

for (i = 0; i < (N / 2); i++)

{

j = N - i - 1;

float t = a[i];

a[i] = a[j];

a[j] = t;

perestanovki += 2;

sravnenia++;

}

free(b);

}