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**

**«Вычисление арифметических выражений»**

**Выполнил**:

студент группы 382003-1

Устинов А.В.

**Проверил**:

ассистент каф. МОСТ,

Волокитин В.Д.

Нижний Новгород

2020

**Содержание**

[Постановка задачи 3](#_Toc26962562)

[Метод решения 4](#_Toc26962563)

[Руководство пользователя 6](#_Toc26962564)

[Описание программной реализации 7](#_Toc26962565)

[Подтверждение корректности 9](#_Toc26962566)

[Результаты экспериментов 10](#_Toc26962567)

[Заключение 13](#_Toc26962568)

[Приложение 14](#_Toc26962569)

# Постановка задачи

Задача состоит в написании программы для сортировки массива вещественных чисел типа float различными алгоритмами: **сортировкой вставками**, **быстрой сортировкой Хоара**, **сортировкой слиянием** и **поразрядной сортировкой**. Также необходимо посчитать количество сравнений и перестановок элементов в каждой сортировке. Требований по размеру массива нет, но требуется проверить работу алгоритмов при различных состояниях его элементов (лучший, средний и худший случаи) так, чтобы показать их теоретическую сложность в каждом случае.

# Метод решения

**1. Сортировка вставками**

Идея состоит в последовательном формировании упорядоченной части массива за счёт вставки очередных элементов на своё место. Изначально отсортированная часть состоит из одного первого элемента, затем последовательно считываются другие элементы массива и размещаются на подходящее место в уже отсортированной части массива, сохраняя при этом относительное расположение одинаковых элементов (является устойчивым).

**2. Быстрая сортировка Хоара**

Алгоритм работает по принципу «разделяй и властвуй»: исходная задача разбивается на более мелкие подзадачи, а общее решение получается посредством комбинирования решений подзадач. Сортировка производится путём рекурсивного разбиения исходного массива на более мелкие части с сопутствующим этому упорядочиванием элементов. В массиве выбирается опорный элемент, относительно которого справа располагаются элементы, большие его, слева – меньшие его. При перестановке элементы в разных частях меняются местами, переставляются элементы на наибольшем расстоянии друг от друга. Затем эта же процедура применяется к каждому подмассиву до тех пор, пока размер подмассива не станет равным 1 (массив из 1 элемента всегда считается упорядоченным). По выполнении всех операций массив будет отсортирован.

**3. Сортировка слиянием**

Ещё один алгоритм типа «разделяй и властвуй». Он базируется на идее слияния двух упорядоченных массивов в один упорядоченный. Путём рекурсии исходный массив разбивается на подмассивы путём деления пополам до тех пор, пока подмассивы не будут состоять из одного элемента. В рекурсивном подъёме происходит слияние двух упорядоченных массивов в один упорядоченный: последовательно сравниваются элементы двух массивов, и меньший заносится в результирующий массив. По окончании одного из массивов оставшиеся элементы другого добавляются в массив-результат.

**4. Поразрядная сортировка**

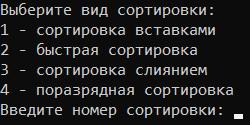
Алгоритм, работающий за линейное время. Он существенно отличается от предыдущих алгоритмов принципом работы, так как не использует сравнения. Является существенной модификацией сортировки подсчётом, в которой подсчитываются все значения элементов исходного массива, а затем на основе подсчёта выводится упорядоченный массив. Идея поразрядной сортировки состоит в сортировке ключей по отдельным разрядам, что приводит к упорядочиванию и всего массива. При работе с числами (целыми или вещественными) каждый разряд можно сортировать устойчивой сортировкой подсчётом, т.к. количество возможных значений разряда невелико (256 при выборе в качестве разряда 1 байт числа). За отсутствие сравнений приходится платить выделением О(n+m) дополнительной памяти, n – размер массива, m – количество различных значений разряда (для чисел это, опять же, 256).

# Руководство пользователя

При запуске программы пользователю будет предложено ввести размер массива (это должно быть целое положительное число):

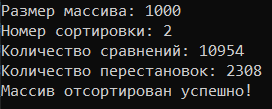
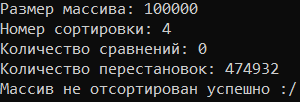


После ввода размера массива появляется возможность выбрать вид сортировки:



При вводе «1» к массиву будет применена сортировка вставками, при вводе «2» - быстрая сортировка, при вводе «3» - сортировка слиянием, при вводе «4» - поразрядная сортировка. Если было введено значение, не совпадающее с описанными выше, будет выведено сообщение о некорректном вводе и будет предложено вновь ввести число.

В случае корректного ввода программа выполняет сортировку, после чего выводит в консоль основные данные и результаты: **длину массива**, **номер выбранной сортировки**, **число сравнений и перестановок** элементов массива во время работы алгоритма и **результат проверки массива на отсортированность** **(«Массив отсортирован успешно!»** в случае правильной работы алгоритма и **«Массив не был отсортирован успешно :/»** при неправильном расположении элементов после сортировки).



Пример правильной работы программы

Пример неправильной работы программы

# Описание программной реализации

Программа состоит из одного файла (**main.cpp**), в котором реализованы все функции: функции сортировок, вспомогательные функции и главная (**main**).

**void insertsort(float\* array, int size) –** функция сортировки вставками. Принимает сортируемый массив и длину этого массива. Состоит из цикла от i=1 до n-1 (проходит по всем элементам массива, кроме нулевого) в который вложен цикл, переставляющий i-й элемент с предыдущими, пока он не достигнет начала массива либо не найдётся меньший его элемент слева.

**void quicksort(float\* array, int l, int r) –** функция быстрой сортировки. Принимает сортируемый массив, левую и правую границы массива. Осуществляет рекурсию, разделяя принимаемый массив на два и вызывая себя для каждой части, пока принимаемый подмассив не достигнет длины 1. Использует вспомогательную функцию **int partition(float\* a, int l, int r),** которая принимает сортируемый подмассив, его левую и правую границы. Она находит средний по расположению элемент массива **v**, потом ищет в первой половине элемент **a[i]**, больший **v** и с минимальным индексом, во второй половине - элемент **a[j]**, меньший **v** и с максимальным индексом, переставляет их. То же повторяется для массива от **i+1** до **j-1**, пока **i** не станет больше **j**. Функция возвращает значение **j**, которое и служит точкой разбиения массива для quicksort.

**void mergesort(float\* array, int n) –** функция сортировки слиянием. Принимает сортируемый массив и его длину. Осуществляет рекурсию, разделяя входящий массив пополам и вызывая себя для каждой половины, пока длина входящего массива не станет равна 1. Непосредственно сортировку осуществляет функция **void merge(float\* arr, float\* l, float\* r, int lcount, int rcount),** принимающая массив, в который будут слияться 2 массива, левую и правую половины подмассива и левую и правую границы массива-слияния. Объединение двух упорядоченных массивов в один упорядоченный производится следующим образом: для каждого из трёх массивов заводится счётчик, берётся меньший из двух первых элементов подмассивов и записывается в результирующий массив; счётчики номеров элементов результирующего массива и подмассива, из которого был взят элемент, увеличиваются на 1. Когда один из массивов заканчивается, оставшиеся элементы второго просто переносятся последовательно в массив-слияние.

**void radixSort(float\* in, float\* out, long N)** – функция поразрядной сортировки. Принимает основной массив, дополнительный массив с длиной, равной основному, и длину массива. Использует вспомогательные функции:

**void** **createCounters(float\* data, long\* counters, long N)** для подсчёта значений каждого байта у элементов (принимает исходный массив, массив подсчёта и длину исходного массива) и **void radixPass(short offset, long N, float\* source, float\* dest, long\* count)**, которая производит сортировку одного разряда (принимает номер байта от 0 до 3, длину массива, исходный и дополнительный массивы, указатель на элемент массива подсчёта, начиная с которого будет происходить сортировка по текущему разряду). Как и в устойчивой сортировке подсчётом, массив **counters** преобразуется так, чтобы в каждом сегменте, соответствующем одному байту, элемент равнялся сумме значений предшествующих элементов этого сегмента. После этого происходит сортировка элементов массива по текущему разряду. Функция **radixSort** создаёт массив подсчёта, заполняет его с помощью **createCounters** и вызывает для сортировки каждого разряда чисел в массиве **radixPass.** При наличии в массиве отрицательных чисел по окончании сортировки они будут располагаться справа от неотрицательных в обратном порядке (из-за особенностей их хранения в памяти), поэтому дополнительно производится перестановка элементов массива для получения правильного порядка.

# Подтверждение корректности

Для подтверждения корректности сортировок используется функция

**void CheckArray(float\* arr, float\* check, int n),** которая принимает отсортированный массив, его неотсортированную копию и длину массива. Проверочный массив сортируется библиотечной быстрой сортировкой **qsort**, затем поэлементно сравниваются оба массива. Для этого создаётся и обнуляется счётчик, который увеличивается на 1 при несовпадении элементов. Если по окончании проверки он равен 0, выводится сообщение, что исходный массив отсортирован корректно, в противном случае – сообщение о неуспешной работе сортировки. Пример соответствующих выводов есть в разделе «Руководство пользователя».

# Результаты экспериментов

Примечание: далее под n подразумевается количество элементов в массиве.

**1) Сортировка вставками**

**Сложность алгоритма**: **лучший случай** – О(n), **средний и худший случаи** – O(n2).

**Лучшим случаем** является полностью упорядоченный массив, **худшим случаем** – обратно упорядоченный массив; за **средний случай** будет принят случайно заполненный массив.

Зависимость количества сравнений (сверху) и перестановок (снизу) от n:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Количество элементов | 50 | 100 | 1000 | 2000 | 5000 |
| Лучший случай | 49 | 99 | 999 | 1999 | 4999 |
| Лучший случай | 0 | 0 | 0 | 0 | 0 |
| Средний случай | 579 | 2604 | 254926 | 989021 | 6239697 |
| Средний случай | 534 | 2511 | 253931 | 987030 | 6234707 |
| Худший случай | 1225 | 4950 | 499500 | 1999000 | 12497500 |
| Худший случай | 1225 | 4950 | 499500 | 1999000 | 12497500 |

По таблице видно, что в лучшем случае количество сравнений растёт линейно (О(n)), в среднем и худшем случаях – квадратично (О(n2)), что согласуется с теоретическими данными.

**2) Быстрая сортировка**

**Сложность алгоритма**: **лучший** и **средний случаи** – О(n\*log(n)), **худший случай** – О(n2).

**Лучшим случаем** является отсортированный массив, **средним случаем** – случайно заполненный массив. **Худший случай** крайне трудно реализовать: он возникает, когда опорный элемент – максимальный или минимальный в массиве. При выборе в качестве опорного элемента середины массива очень сложно подобрать входные данные, при которых программа отработает за квадратичное время, из-за чего этот случай исключён из рассмотрения.

Зависимость количества сравнений (сверху) и перестановок (снизу) от n:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Количество элементов | 50 | 100 | 1000 | 10000 | 100000 |
| Лучший случай | 237 | 573 | 8977 | 123617 | 1568929 |
| Лучший случай | 0 | 0 | 0 | 0 | 0 |
| Средний случай | 316 | 611 | 11660 | 137736 | 1662906 |
| Средний случай | 61 | 148 | 2278 | 31258 | 412637 |

По графикам можно сделать вывод, что результаты эксперимента совпадают с теоретической оценкой О(n\*log(n)).

**3) Сортировка слиянием**

**Сложность алгоритма**: О(n\*log(n)) **во всех случаях**.

**Лучшим случаем** выступает отсортированный массив, **средним случаем** – случайно заполненный массив. **Худший случай** крайне сложно реализовать, и он не рассматривается.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Количество элементов | 50 | 100 | 1000 | 2000 | 5000 |
| Лучший случай | 133 | 316 | 4932 | 10864 | 29804 |
| Лучший случай | 286 | 672 | 9976 | 21952 | 61808 |
| Средний случай | 219 | 542 | 8678 | 19401 | 55243 |
| Средний случай | 286 | 672 | 9976 | 21952 | 61808 |

Зависимость количества сравнений (сверху) и перестановок (снизу) от n:

# Графики подтверждают теоретическую сложность сортировки О(n\*log(n)).

**4) Поразрядная сортировка**

**Сложность алгоритма**: О(n) **во всех случаях** (считаются перестановки элементов массива).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Количество элементов | 50 | 100 | 1000 | 10000 | 100000 |
| Количество сравнений | 0 | 0 | 0 | 0 | 0 |
| Количество перестановок | 250 | 500 | 5000 | 50000 | 500000 |

Ввиду не зависящего от входящих данных принципа работы нет смысла рассматривать каждый случай в отдельности; будет рассматриваться результат при случайно заполненном массиве.

Нетрудно заметить, что число перестановок растёт линейно относительно n, что совпадает с теоретической сложностью.

**Заключение**

Итак, поставленная в начале работы задача выполнена: написана программа для исполнения заданных сортировок (код см. в Приложении), описаны взаимодействие пользователя с программой, структура проекта, способ проверки корректности алгоритмов, приведены результаты экспериментов, согласующиеся с теоретическими данными.

**Приложение**

Код сортировок:

void insertsort(float\* array, int size) /\*Сортировка вставками\*/

{

int k = 0;

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

{

k = i;

for (int j = i; (j > 0) && (array[j - 1] > array[j]); j--)

{

if (array[j - 1] > array[j])

{

float tmp = array[j - 1];

array[j - 1] = array[j];

array[j] = tmp;

perm++;

k = j - 1;

}

}

if (k != 0) comp++;

}

comp += perm;

}

int partition(float\* a, int l, int r) /\*Функция перестановки элементов в быстрой сортировке\*/

{

float v = a[(l + r) / 2];

int i = l;

int j = r;

while (i <= j)

{

while (a[i] < v) { i++; comp++; }

while (a[j] > v) { j--; comp++; }

if (i >= j) break;

swap(&a[i++], &a[j--]);

perm++;

comp++;

}

return j;

}

void quicksort(float\* array, int l, int r) /\*Быстрая ортировка\*/

{

if (l < r)

{

int q = partition(array, l, r);

quicksort(array, l, q);

quicksort(array, q + 1, r);

}

}

void merge(float\* arr, float\* l, float\* r, int lcount, int rcount) /\*Функция слияния двух упорядоченных массивов\*/

{

int i = 0, j = 0, k = 0;

while ((i < lcount) && (j < rcount))

{

comp++;

perm++;

if (l[i] < r[j]) arr[k++] = l[i++];

else arr[k++] = r[j++];

}

while (i < lcount) {arr[k++] = l[i++]; perm++;}

while (j < rcount) {arr[k++] = r[j++]; perm++;}

}

void mergesort(float\* array, int n) /\*Сортировка слиянием\*/

{

int mid;

if (n < 2) return;

mid = n / 2;

float\* l = (float\*)malloc(mid \* sizeof(float));

float\* r = (float\*)malloc((n - mid) \* sizeof(float));

for (int i = 0; i < mid; i++) l[i] = array[i];

for (int i = mid; i < n; i++) r[i - mid] = array[i];

mergesort(l, mid);

mergesort(r, n - mid);

merge(array, l, r, mid, n - mid);

free(l);

free(r);

}

void createCounters(float\* data, long\* counters, long N) /\*Подсчёт значений байтов (radix sort)\*/

{

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

unsigned char\* dataEnd = (unsigned char\*)(data + N);

unsigned short i;

memset(counters, 0, 256 \* sizeof(float) \* sizeof(long));

while (bp != dataEnd) {

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

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

}

}

void radixPass(short offset, long N, float\* source, float\* dest, long\* count) /\*Сортировка массива по одному разряду (radix sort)\*/

{

float\* sp;

long s = 0, c, i, \* cp = count;

unsigned char\* bp;

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

c = \*cp; \*cp = s; s += c;

}

bp = (unsigned char\*)source + offset;

sp = source;

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

{

cp = count + \*bp;

dest[\*cp] = \*sp;

(\*cp)++;

perm++;

}

}

void radixSort(float\* in, float\* out, long N) /\*Поразрядная сортировка\*/

{

long\* count;

long\* counters = (long\*)malloc(1024 \* sizeof(long));

long k = 0;

createCounters(in, counters, N);

for (unsigned short i = 0; i < sizeof(float); i++) {

count = counters + 256 \* i;

radixPass(i, N, in, out, count);

for (long j = 0; j < N; j++)

in[j] = out[j];

}

while (in[k] >= 0 && k < N && !(k > 0 && in[k] <= 0 && in[k - 1] > 0)) k++;

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

in[i] = out[N-1-i];

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

in[N - k + i] = out[i];

perm += N;

free(counters);

}