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**:

студентка группы 382003-1

Шамонина А.С.

**Проверил**:

ассистент каф. МОСТ,

ВолокитинВ.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

Требовалось написать программу, содержащую четыре вида сортировки массива данных типа float: сортировку пузырьком, быструю сортировку, сортировку слиянием и поразрядную сортировку. На вход должна подаваться последовательность, состоящая из n чисел (a1, a2, …, an), на выходе должна быть перестановка входной последовательности таким образом, что a1 ≤ a2≤ … ≤ an

# Метод решения

1. **Сортировка пузырьком**

*Идея алгоритма:* Пузырек воздуха в стакане воды поднимается со дна вверх.

*Алгоритм:* Состоит из повторяющихся проходов по сортируемому массиву. Элементы последовательно сравниваются попарно каждый проход и, если порядок в паре неверный, выполняется обмен элементов. Проходы по массиву повторяются {\displaystyle N-1}до тех пор, пока на каком либо из проходов не окажется, что обмены не происходят, это означает, что массив отсортирован. При каждом проходе алгоритма по внутреннему циклу, очередной наибольший элемент массива ставится на своё место в конце массива рядом с предыдущим «наибольшим элементом», а наименьший элемент перемещается на одну позицию к началу массива. Количество сравнений в каждом проходе равно m-i, где i – это номер прохода по массиву (первый, второй, третий и т.д.).

1. **Быстрая сортировка**

*Идея алгоритма:* «разделяй и властвуй»

*Алгоритм:* Из исходного массива выбирается некоторый элемент, который принимается в качестве разделителя или опорного элемента(ключа). Все ключи, меньшие разделителя, располагаются до него, а все большие после . Перестановка элементов выполняется  путём обмена местами ключей,  которые необходимо переместить в  другую часть массива. При этом обмениваются ключи,  расположенные на большом  расстоянии друг от друга и этим  достигается высокий эффект  упорядочивания.

Массив a[l…r] типа T(в нашем случае типа float) разбивается на два подмассива a[l…q] и a[q+1…r], таких, что каждый элемент a[l…q] меньше или равен a[q], который в свою очередь, не превышает любой элемент подмассива a[q+1…r]. Индекс вычисляется в ходе процедуры разбиения. Подмассивы a[l…q] и a[q+1…r] сортируются с помощью рекурсивного вызова процедуры быстрой сортировки

1. **Сортировка слиянием**

*Идея алгоритма:* объединение двух отсортированных массивов

*Алгоритм:* Сортируемый массив разбивается на две части примерно одинакового размера. Каждая из получившихся частей сортируется отдельно, в нашем случае тем же самым алгоритмом. Два упорядоченных массива половинного размера соединяются в один. Рекурсивное разбиение задачи на меньшие происходит до тех пор, пока размер массива не достигнет единицы (любой массив длины 1 можно считать упорядоченным). 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.

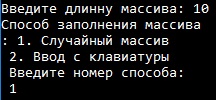
1. **Поразрядная сортировка**

*Идея алгоритма:* Каждый ключ сортировки рассматривается как k-значное число, каждая цифра которого находится в диапазоне от 0 до m-1. Поочередно используется устойчивая сортировка (в нашем случае сортировка подсчетом) для каждой цифры справа налево.

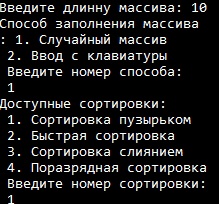
*Алгоритм:* Сравнение элементов производится поразрядно: сначала сравниваются значения одного крайнего разряда, и элементы группируются по результатам этого сравнения, затем сравниваются значения следующего разряда, соседнего, и элементы либо упорядочиваются по результатам сравнения значений этого разряда внутри образованных на предыдущем проходе групп, либо переупорядочиваются в целом, но сохраняя относительный порядок, достигнутый при предыдущей сортировке. Затем аналогично делается для следующего разряда, и так до конца.

# Руководство пользователя

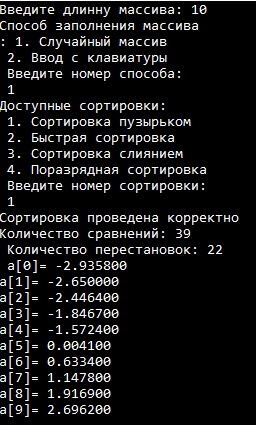
При запуске программы появляется окно ввода, в котором предлагается ввести длину массива, который пользователь собирается отсортировать. Далее появится меню, в котором можно выбрать способ заполнения массива.



Затем, после ввода длинны, и выбора способа заполнения массива появляется меню, в котором можно выбрать вид сортировки на усмотрение пользователя. Программа просит ввести порядковый номер нужной пользователю сортировки.



В результате работы программы будет выведено количество перестановок и сравнений и результат проверки отсортированного массива на корректность. Так же выведется отсортированный массив.

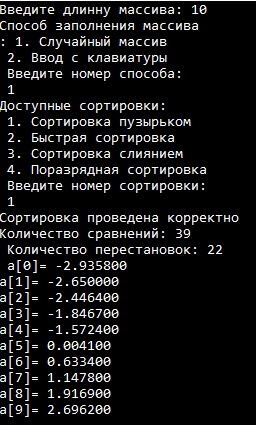
****

# Описание программной реализации

|  |  |  |
| --- | --- | --- |
| Список функций | Аргументы | Что делает |
| void bubble\_sort | Указатель на массив(float), размер массива | Сортирует массив по возрастанию, сравнивая и переставляя элементы, начиная с конца. Ничего не возвращает |
| void quicksort | Указатель на массив (float), номер крайнего левого и крайнего правого элемента | Делит массив на две части, серединный элемент выбирает опорным, относительно него отдельно сортирует части массива. Ничего не возвращает |
| void merge | Указатель на изначальный и дополнительный массивы(float), номер крайнего правого и левого элемента, номер элемента из середины массива | Отвечает за упорядочение частей массива, после чего собирает их в один . Ничего не возвращает. |
| void mergesort | Указатель на изначальный и дополнительный массивы(float), номер крайнего правого и левого элемента | Обеспечивает рекурсивное разбиение задачи, за счет чего части массива сортируются отдельно. Ничего не возвращает |
| void createCounters | Указатель на исходный массив(float), указатель на дополнительный массив с ячейками типа long, размер исходного массива | Создает ячейки для возможных значений разрядов чисел. Ничего не возвращает. |
| void radixPass | Номер байта Offset, число элементов , исходный массив массив, куда будут записываться числа, отсортированные по байту Offset, массив счетчиков count, соответствующий текущему проходу | На первом проходе подсчитывается количество различных значений в этом разряде. Для каждого возможного значения подготавливаются массивы. При втором проходе выписываются сами элементы в эти массивы. Второй и последующие проходы выполняются отдельно над каждой корзиной, полученной на предыдущем проходе, с делением её на «подкорзины» и сравнением соответственно второго и последующих символов строк. |
| void signedradixLastPass | Номер байта Offset, число элементов , исходный массив массив, куда будут записываться числа, отсортированные по байту Offset, массив счетчиков count, соответствующий текущему проходу | Работает со старшими байтами, обеспечивая правильное положение знаковых чисел в отсортированном массиве. |
| void radixSort | Указатель на исходный массив(float), указатель на дополнительный массив, размер массива. | Осуществляет sizeof(float) проходы по направлению от младшего байта к старшему |

# Подтверждение корректности

Для подтверждения корректности отсортированного массива в программе используется вызов функции checking в которой производится вызов библиотечной функции qsort, с помощью неё сортируется копия исходного массива. Затем копия сравнивается с массивом после выбранной пользователем сортировки.



Если сортировка пройдет корректно, будет произведен вывод фразы «Сортировка работает корректно», в худшем случае «Сортировка проведена некорректно»

# Результаты экспериментов

1. Сортировка пузырьком

**Сложность алгоритма**: 0(n) в лучшем случае(отсортированный массив), 0(n^2) в среднем(случайные числа) и худшем(массив в порядке убывания).

Из таблицы можно сделать вывод, что число сравнений растет линейно в лучшем случае и квадратично в худшем и среднем, что подтверждает теоретические данные.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Сравнения/Размер массива | 100 | 300 | 500 | 700 | 1000 |
| Худший случай | 4950 | 44850 | 124750 | 244650 | 499500 |
| Случайные числа | 4922 | 44640 | 122670 | 244154 | 498905 |
| Лучший случай | 99 | 299 | 499 | 699 | 999 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Перестановки/Размер массива | 100 | 300 | 500 | 700 | 1000 |
| Худший случай | 4950 | 44850 | 124750 | 244650 | 499500 |
| Случайные числа | 2409 | 23060 | 61313 | 122258 | 251685 |
| Лучший случай | 0 | 0 | 0 | 0 | 0 |

1. Быстрая сортировка

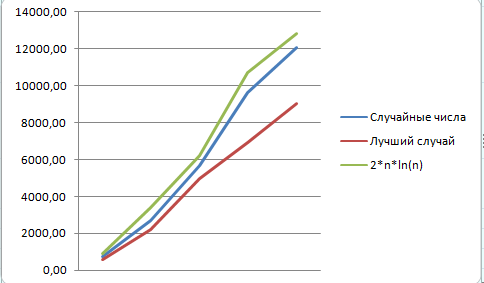
**Сложность алгоритма**: 0(nlogn) в лучшем случае(отсортированный массив и в среднем(случайные числа) и), 0(n^2) в худшем(массив в порядке убывания). Худший случай может возникать редко, для его реализации понадобилось бы найти максимальный или минимальный элемент, чтобы использовать его в качестве опорного. Поэтому в таблице он не рассматривается.

В лучшем случае перестановки производятся из-за перестановки ведущего элемента.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Перестановки/Размер массива | 100 | 300 | 500 | 800 | 1000 |
| Случайные числа | 168 | 559 | 1167 | 2033 | 2288 |
| Лучший случай | 63 | 172 | 257 | 511 | 515 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Сравнения/Размер массива | 100 | 300 | 500 | 800 | 1000 |
| Случайные числа | 593 | 2147 | 4486 | 7564 | 9751 |
| Лучший случай | 543 | 2070 | 4732 | 6700 | 8514 |

Из таблицы и графика можно сделать вывод, что сложность совпадает с теоретической

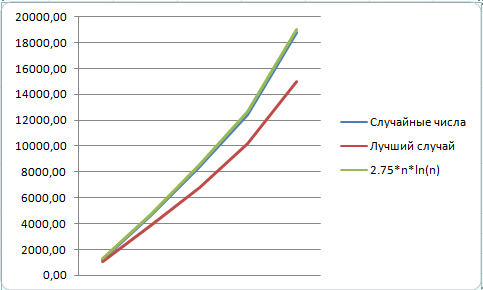
1. 
2. Сортировка слиянием

**Сложность алгоритма**: 0(nlogn) во всех случаях. Худших случаев почти не имеет, поэтому рассмотрим случайный массив и отсортированный

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Сравнения/Размер массива | 100 | 300 | 500 | 700 | 1000 |
| Случайные числа | 552 | 2137 | 3924 | 5817 | 8832 |
| Лучший случай | 356 | 1308 | 2272 | 3504 | 5044 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Перестановки/Размер массива | 100 | 300 | 500 | 700 | 1000 |
| Случайные числа | 672 | 2488 | 4488 | 6676 | 9976 |
| Лучший случай | 672 | 2488 | 4488 | 6676 | 9976 |

Из таблицы и графика можно сделать вывод, что сложность совпадает с теоретической



1. Сортировка поразрядная

**Сложность алгоритма**: 0(n) во всех случаях, так как работа алгоритма не зависит от вводимых данных, поэтому рассмотрим случайно заполненный массив.

Из таблицы можно сделать вывод, что сложность совпадает с теоретической, так как число перестановок зависит линейно от размера массива

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Размер массива | 100 | 300 | 500 | 700 | 1000 |
| Сравнения | 0 | 0 | 0 | 0 | 0 |
| Перестановки | 400 | 1200 | 2000 | 2800 | 4000 |

# Заключение

В ходе работы была написана программа, сортирующая массив данных типа float 4 видами сортировок. Так же были описаны руководство пользователя и способ проверки результата сортировки на корректность. Были приведены результаты экспериментов, подтверждающие теорию.

# Приложение

void bubble\_sort(float \*array, const size\_t size)

{

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

{

flag = 0;

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

{

count1++;

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

{

count2++;

float tmp = array[j];

array[j] = array[j + 1];

array[j + 1] = tmp;

flag = 1;

}

}

if (flag == 0)

break;

}

}

void quicksort (float \*a, int L, int R)

{

int i=L,j=R;

float q=a[(L+R)/2];

do {

while (a[i] < q) { i++; count1++; }

while (a[j] > q) { j--; count1++; }

if (i<=j)

{

float w=a[i];

a[i]=a[j];

a[j]=w;

i++; j--; count1++; count2++;

}

}

while (i<=j);

if (L<j)

quicksort(a,L,j);

if (i<R)

quicksort(a,i,R);

}

void merge(float\* array,float\* second, int l, int q, int r)

{ int h, i, j, k;

q = (l + r) / 2; h = l; i = l; j = q + 1;

while ((h <= q) && (j <= r))

{ count1++;

if (array[h] <= array[j])

{

count2++;

second[i] = array[h];

h++;

}

else

{

count2++;

second[i] = array[j];

j++;

}

i++;

} if (h > q)

{ for (k = j; k <= r; k++)

{ count2++;

second[i] = array[k];

i++;

}

}

else

{ for (k = h; k <= q; k++)

{ count2++;

second[i] = array[k];

i++;

}

} for (k = l; k <= r; k++)

{ array[k] = second[k];

}

}

void mergesort(float \*array, float \*second, int l, int r) {

if (l < r) {

int q = (l + r) / 2;

mergesort(array, second, l, q);

mergesort(array,second , q + 1, r);

merge(array,second, l, q, r);

}

}

void createCounters(float\* in, long\* counters, long N) {

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

unsigned char\* end = (unsigned char\*)(in + N);

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

while (bp != end) {

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

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

}

}

}

void radixPass(short Offset, long N, float\* in, float\* out, long\* count) {

float\* sp;

long s, c, \* cp;

unsigned char\* bp;

s = 0;

cp = count;

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

c = \*cp;

\*cp = s;

s += c;

}

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

sp = in;

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

cp = count + \*bp;

out[\*cp] = \*sp;

(\*cp)++;

count2++;

}

}

void signedradixLastPass(short Offset, long N, float\* in, float\* out, long\* count) {

float\* sp;

long s, c, \* cp, numNeg = 0;

unsigned char\* bp;

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

s = numNeg;

cp = count;

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

c = \*cp;

\*cp = s;

s += c;

}

cp = count + 255;

s = 0; \*cp = 0;

for (int i = 255; i >= 128; i--, cp--) {

s += \*cp;

\*cp = s;

}

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

sp = in;

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

cp = count + \*bp;

if (\*bp < 128) {

out[\*cp] = \*sp;

(\*cp)++;

}

else {

(\*cp)--;

out[\*cp] = \*sp;

}

count2++;

}

}

void radixSort(float\* in, float\* out, int N) {

int i;

long\* counters = (long\*)malloc(sizeof(float) \* 256 \* sizeof(long));

long\* count;

createCounters((float\*)in, counters, N);

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

count = counters + 256 \* i;

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

float\* tmp = in;

in = out;

out = tmp;

}

count = counters + 256 \* (i);

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

}