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»**

**Выполнил**:

студент группы 382003-1

Медведев М. С.

**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

Целью этой работы было научиться писать программу, работающую с массивами, элементы которых имеют тип double. Моей задачей являлось реализовать 4 сортировки: сортировка простыми обменами (сортировка «пузырьком»), сортировка расчёской, сортировка слиянием, поразрядная сортировка. Программа должна корректно работать и быть понятной для пользователя. Помимо написания сортировок требовалось замерить количество перестановок элементов в массиве и их перестановок, выполненное при работе программы.

# Метод решения

Чтобы решить поставленную задачу, нужно знать алгоритмы сортировок.

Алгоритм сортировки «пузырьком» следующий: нужно обходить массив от начала до конца, сравнивая и переставляя неотсортированные соседние элементы друг с другом. В результате первого такого обхода максимальный по значению элемент встанет на последнее место в массиве («всплывет» на последнее место). Затем нужно повторить эту же операцию, в результате которой второй по величине элемент встанет на предпоследнее место. Далее нужно повторять обходы, перемещая максимальные элементы в конец неотсортированной части массива, до тех пор, пока он не будет отсортирован.

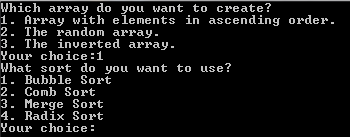
Основная идея сортировки «расчёской состоит в том, чтобы изначально брать максимально большое расстояние между сравниваемыми элементами и по мере упорядочивания массива уменьшать это расстояние до минимального с помощью фактора уменьшения, приблизительно равного 1,247. Таким образом мы как бы расчесываем массив, постепенно разглаживая его. Изначально расстояние между сравниваемыми элементами равно размеру массива, делённое на фактор уменьшения. Затем мы проходим весь массив с этим шагом (очевидно, что с этим шагом больше никаких сравнений и перестановок не будет, так как сравнивать и переставлять не с чем) и еще раз делим шаг на фактор уменьшения, проходя снова по массиву. Так происходит до тех пор, пока разность индексов не достигнет единицы (сравниваться уже будут соседние элементы). После этого массив сортируется описанной ранее сортировкой «пузырьком». Благодаря эти действиям пользователь получает отсортированный массив.

Алгоритм сортировки слиянием немного сложнее двух предыдущих. Его можно разделить на три этапа. Суть первого шага состоит в том, что сортируемый массив рекурсивно разбивается на две части примерно одинакового размера до тех пор, пока размер его размер не будет равным единице. Массив из одного элемента можно считать упорядоченным. Затем наступает второй шаг: разделив массив на две части, нужно их отсортировать. Также нужно каждую часть поделить на две части и отсортировать и т.д. Третьим шагом является слияние двух упорядоченных массивов. Сначала соединяются массивы длины 1 в массивы длины 2, затем массивы длины 2 в массивы длины 4 и т.д. В общем случае слияние подмассивов происходит так: на каждом шаге мы берём меньший из двух первых элементов подмассивов и ставим его в начало результирующего массива. Когда один подмассив заканчивается, мы добавляем его в конец результирующего подмассива.

В поразрядной сортировке всё происходит немного по-другому. Сравнения производятся поразрядно. Изначально сравниваются значения крайнего разряда, в результате этого элементы группируются по результатам этого сравнения. Затем сравниваются значения разряда, стоящего левее предыдущего, после чего элементы либо упорядочиваются по результатам значений этого разряда внутри образованных на предыдущем проходе групп, либо переупорядочиваются совсем, при этом сохраняя порядок, достигнутый при предыдущей сортировке. Аналогично делается для следующего разряда, и так до конца сортировки. Существует два варианта этой сортировки: LSD (сортировка с выравниванием по младшему разряду, направо, к единицам) и MSD (сортировка с выравниванием по старшему разряду). В своей работе я использовал первый вариант, где сначала сравниваются единицы, затем десятки, сотни и т.д.

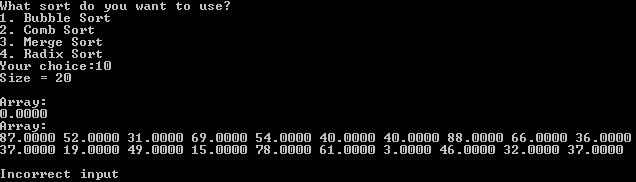
# Руководство пользователя

При запуске программы пользователю будет предложено выбрать массив, который он хочет создать, и сортировку, которая будет работать с массивом:

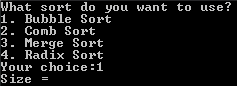


В поле «Your choice» пользователь должен ввести номер сортировки и массива, которые хочет использовать.

В случае некорректного ввода будет выведена следующая фраза:

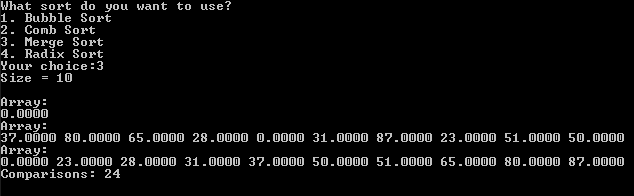


После этого в поле «Size» пользователь должен будет ввести размер массива, который нужно отсортировать:

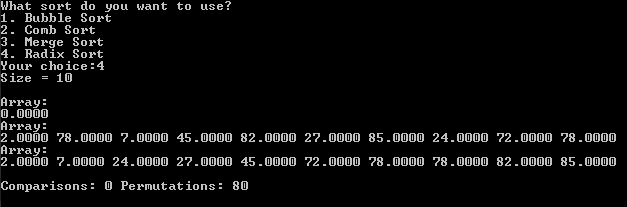


На этом свобода действий пользователя заканчивается. В консоли будет выведен случайно сгенерированный массив. Строчкой ниже будет выведен отсортированный массив. Еще ниже будет выведено количество сравнений и перестановок.

В сортировке слиянием нет перестановок, поэтому выведется только количество сравнений:



В поразрядной сортировке не происходит сравнение элементов, поэтому оно всегда будет равно нулю:



Если сортировка всё сделала правильно, то будет выведено следующее:



После этого пользователь может закрыть консоль нажатием любой клавиши.

# Описание программной реализации

Основные функции:

void random(double\* arr, size\_t size).

Аргументы этой функции: arr (сортируемый массив), size (размер массива).

Тип возвращаемого значения: void.

Имя функции: random.

Эта функция создает случайный массив.

void bubble\_sort(double\* arr, size\_t size).

Аргументы этой функции: arr (сортируемый массив), size (размер массива).

Тип возвращаемого значения: void.

Имя функции: bubble\_sort.

Эта функция сортирует массив, используя алгоритм сортировки «пузырьком».

void comb\_sort(double\* arr, size\_t size).

Аргументы этой функции: arr (сортируемый массив), size (размер массива).

Тип возвращаемого значения: void.

Имя функции: comb\_sort.

Эта функция сортирует массив, используя алгоритм сортировки расчёской.

void output(double\* arr, size\_t size).

Аргументы этой функции: arr (сортируемый массив), size (размер массива).

Тип возвращаемого значения: void.

Имя функции: output.

Эта функция выводит сгенерированный и отсортированный массив на экран.

void merge(double\* arr, double\* scndarr, int scndCount, double\* thrarr, int thrCount).

Аргументы этой функции: arr (сортируемый массив), scndarr (левый подмассив), scndCount (количество элементов в левом подмассиве) , thrarr (правый подмассив), thrCount (количество элементов в правом подмассиве).

Тип возвращаемого значения: void.

Имя функции: merge.

Эта функция производит слияние двух массивов в один.

void merge\_sort(double\* arr, size\_t size).

Аргументы этой функции: arr (сортируемый массив), size (размер массива).

Тип возвращаемого значения: void.

Имя функции: merge\_sort.

Эта функция сортирует массив, используя алгоритм сортировки слиянием.

void counter(int cmp, int c).

Аргументы этой функции: cmp (количество сравнений), c (количество перестановок).

Тип возвращаемого значения: void.

Имя функции: counter.

Эта функция выводит показатели сравнений и перестановок на экран.

void comp(int cmp).

Аргументы этой функции: cmp (количество сравнений).

Тип возвращаемого значения: void.

Имя функции: comp.

Эта функция выводит показатель сравнений для сортировки слиянием на экран.

void create\_counters(double\* data, long\* counters, long N);

Аргументы этой функции: data (сортируемый массив), counters (массив для счётчиков), N (число элементов).

Тип возвращаемого значения: void.

Имя функции: create\_counters.

Эта функция создает счётчики.

void radix\_pass(short Offset, long N, double\* source, double\* dest, long\* count).

Аргументы этой функции: Offset (номер байта), N (число элементов), source (исходный массив), dest (массив, в который будут записываться элементы, отсортированные по байту Offset), count.

Тип возвращаемого значения: void.

Имя функции: radix\_pass.

Эта функция записывает числа в массив dest, отсортированные по байту Offset.

void radix\_sort(double\* in, double\* out, long\* counters, long N).

Аргументы этой функции: in(сортируемый массив), out, counters, N(количество элементов).

Тип возвращаемого значения: void.

Имя функции: radix\_sort.

Эта функция сортирует массив in из N элементов.

# Подтверждение корректности

Для подтверждения корректности в программе реализована функция

void check(double\* arr, size\_t size). Она имеет аргументы arr, size. Тип возвращаемого значения: void. Она работает так: если предыдущий элемент в отсортированном массиве больше следующего, то на экран выводится надпись «Sort is incorrect». Если же сортировка выполнила свою работу правильно, то выводится надпись «Sort is correct».

# Результаты экспериментов

1) Сортировка «пузырьком».

Теоретическая сложность алгоритма: худший случай – О(n^2), средний случай – О(n^2), лучший случай – О(n).

Лучшим случаем является уже отсортированный массив. Средним случаем является случайно сгенерированный массив. Худшим – отсортированный в обратном порядке массив.

В таблице приведена зависимость количества перестановок и сравнений от размера массива.

|  |  |  |  |
| --- | --- | --- | --- |
|  | 100 | 2000 | 50000 |
| Лучший случай  (сравнения) | 99 | 1999 | 49999 |
| Лучший случай  (перестановки) | 0 | 0 | 0 |
| Средний случай  (сравнения) | 4884 | 1995084 | 1249854705 |
| Средний случай  (перестановки) | 2377 | 979478 | 619770595 |
| Худший случай  (сравнения) | 4950 | 1998700 | 1249838497 |
| Худший случай  (перестановки) | 4909 | 1979094 | 1237473355 |

По результатам экспериментов видно, что показатели совпадают с теоретической сложностью сортировки.

2) Сортировка расчёской.

Лучшим случаем для нее теоретическая сложность равна О(n\*logn), когда массив отсортирован. Средним: О(n^2/n^p), p – количество приращений. Худший случай для неё: О(n^2).

В таблице приведена зависимость количества перестановок и сравнений от размера массива.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 100 | 2000 | 50000 | 100000 |
| Лучший случай  (сравнения) | 1223 | 49994 | 1997961 | 4295797 |
| Лучший случай  (перестановки) | 0 | 0 | 0 | 0 |
| Средний случай  (сравнения) | 1233 | 49994 | 1997961 | 4295797 |
| Средний случай  (перестановки) | 264 | 5691 | 145099 | 290590 |

Исходя из таблицы можно сделать вывод о том, что теоретическая сложность алгоритма совпадает с показаниями, полученными на практике.

3) Сортировка слиянием.

Лучшее случай О(n\*log(n)) обычно, O(n), на упорядоченном массиве. Худшее и среднее время: О(n\*log(n)).

В таблице приведена зависимость сравнений в лучшем и среднем случаях от размера массива.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 100 | 2000 | 50000 | 100000 |
| Лучший случай  (сравнения) | 348 | 12847 | 451851 | 952655 |
| Средний случай  (сравнения) | 543 | 19354 | 716559 | 1532718 |

По данным таблицы видно, что теоретическая сложность алгоритма действительно совпадает с результатами экспериментов.

4) Поразрядная сортировка.

Теоретическая сложность: О(k\*(n+m)), где k – разрядность, n - количество элементов, m - количество значений разряда.

Лучший, средний и худший случаи реализовать нельзя, потому что при фиксированном n всегда выполняется одно и то же количество операций вне зависимости от того, какой массив.

# Заключение

В результате этой работы я смог реализовать сортировки, корректно работающие с одномерными массивами. Также я замерил количество перестановок и сравнений в каждой из них и убедился во временной сложности. Проведенные эксперименты показали, что теоретическая сложность, описанная в различных источниках, совпадает с моими результатами.

# Приложение

void random(double\* arr, size\_t size)

{

int i;

printf("\nArray:\n");

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

arr[i] = rand() % 109 + (-37);

printf("%.4lf ", arr[i]);

}

void check(double\* arr, size\_t size)

{

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

{

if (arr[i - 1] > arr[i])

{

printf("\nSort is incorrect\n");

break;

}

else

{

printf("\nSort is correct\n");

break;

}

}

}

void output(double\* arr, size\_t size)

{

printf("\nArray:\n");

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

printf("%.4lf ", arr[i]);

}

void counter(int cmp, int c)

{

printf("\n\nComparisons: %d Permutations: %d\n\n", cmp, c);

}

void comp(int cmp)

{

printf("\nComparisons: %d\n\n", cmp);

}

void bubble\_sort1(double\* arr, size\_t size)

{

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

{

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

{

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

{

double tmp = arr[j - 1];

arr[j - 1] = arr[j];

arr[j] = tmp;

}

}

}

}

void bubble\_sort2(double\* arr, size\_t size)

{

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

{

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

{

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

{

double tmp = arr[j - 1];

arr[j - 1] = arr[j];

arr[j] = tmp;

}

}

}

}

void bubble\_sort(double\* arr, size\_t size)

{

int flag = 1;

for (size\_t i = 0; i < size - 1 && flag; i++)

{

flag = 0;

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

{

cmp++;

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

{

c++;

double tmp = arr[j - 1];

arr[j - 1] = arr[j];

arr[j] = tmp;

flag = 1;

}

}

}

}

void comb\_sort(double\* arr, size\_t size)

{

int step = size - 1;

double factor = 1.2473309;

while (step >= 1)

{

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

{

cmp++;

if (arr[i] > arr[i + step])

{

c++;

double tmp = arr[i];

arr[i] = arr[i + step];

arr[i + step] = tmp;

}

}

step /= factor;

}

}

void merge(double\* arr, double\* scndarr, int scndCount, double\* thrarr, int thrCount)

{

int i, j, k;

i = 0;

j = 0;

k = 0;

while (i < scndCount && j < thrCount)

{

cmp++;

if (scndarr[i] < thrarr[j])

{

arr[k] = scndarr[i];

k++;

i++;

}

else

{

arr[k] = thrarr[j];

k++;

j++;

}

}

while (i < scndCount)

{

arr[k] = scndarr[i];

k++;

i++;

}

while (j < thrCount)

{

arr[k] = thrarr[j];

k++;

j++;

}

}

void merge\_sort(double\* arr, size\_t size)

{

int mid, i;

double\* scndarr, \* thrarr;

if (size < 2)

{

return;

}

mid = size / 2;

scndarr = (double\*)malloc(mid \* sizeof(double));

thrarr = (double\*)malloc((size - mid) \* sizeof(double));

for (i = 0; i < mid; i++)

{

scndarr[i] = arr[i];

}

for (i = mid; i < size; i++)

{

thrarr[i - mid] = arr[i];

}

merge\_sort(scndarr, mid);

merge\_sort(thrarr, size - mid);

merge(arr, scndarr, mid, thrarr, size - mid);

free(scndarr);

free(thrarr);

}

void create\_counters(double\* data, long\* counters, long N)

{

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

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

unsigned short i;

for (int i = 0; i < 256 \* sizeof(double); i++)

counters[i] = 0;

while (bp != dataEnd)

{

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

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

}

}

void radix\_pass(short Offset, long N, double\* source, double\* dest, long\* count)

{

double\* sp;

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

unsigned char\* bp;

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

{

d = \*cp; \*cp = s; s += d;

}

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

sp = source;

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

{

cp = count + \*bp; dest[\*cp] = \*sp; (\*cp)++;

}

c += N;

}

void radix\_sort(double\* in, double\* out, long\* counters, long N)

{

long\* count;

unsigned short i;

int k = 0;

create\_counters(in, counters, N);

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

{

count = counters + 256 \* i;

radix\_pass(i, N, in, out, count);

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

in[j] = out[j];

}

while (in[k] >= 0 && k < N)

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];

}

free(arr);

free(counters);

}

Эту часть кода я считаю основной, так как здесь описывается то, как устроены функции, какие в них переменные и аргументы.