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

Мосягин К. М.

**Проверил**:

ассистент каф. МОСТ,

Волокитин В. Д.

Нижний Новгород

2020

**Содержание**

Постановка задачи…….…….…….…….…….…….…….….…….….…….…….….3   
Метод решения….…….…….…….…….…….…….…….…….…….…….…….…..4   
Руководство пользователя….…….…….…….…….…….…….…….…….…….…..7  
Описание программной реализации….…….…….…….…….…….…….…….……9   
Подтверждение корректности….…….…….…….…….…….…….…….…….……10   
Результаты экспериментов….…….…….…….…….…….…….…….…….…….…11   
Заключение….…….…….…….…….…….…….…….…….…….…….…….……....13   
Приложение….…….…….…….…….…….…….…….…….…….…….…….………14

# Постановка задачи

Задачей лабораторной работы являлось написание программы, сортирующей массив чисел типа double при помощи:

1. Сортировка пузырьком.
2. Сортировка Шелла.
3. Сортировка слиянием
4. Поразрядной сортировки.

Массив может вводиться пользователем, или задаваться случайными числами. После работы программы массив должен быть отсортирован от меньшего элемента к большему. Так же должна выводиться сложность сортировки, а именно число перестановок и число сравнений.

Стр. 3

# Метод решения

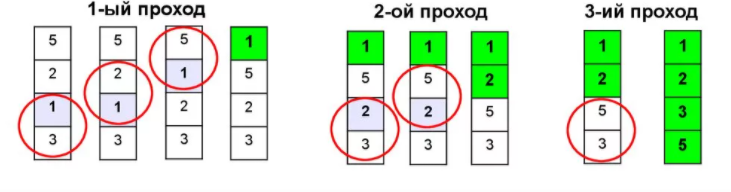
**Сортировка пузырьком (с флагом).**

Идея – пузырек воздуха в стакане воды поднимается со дна вверх.

Сортировка массива осуществляется путем последовательного сравнения и обмена соседних элементов, если предшествующий оказывается больше последующего.

В процессе выполнения данного алгоритма элементы с большими значениями оказываются в конце массива, а элементы с меньшими значениями постепенно перемещаются по направлению к началу массива.

Если при выполнении метода пузырька не было обменов, массив уже отсортирован и остальные проходы не нужны. Следовательно можно ввести переменную flag, показывающую, был ли этот обмен или нет. Если она равна 0, то алгоритм заканчивает работу.

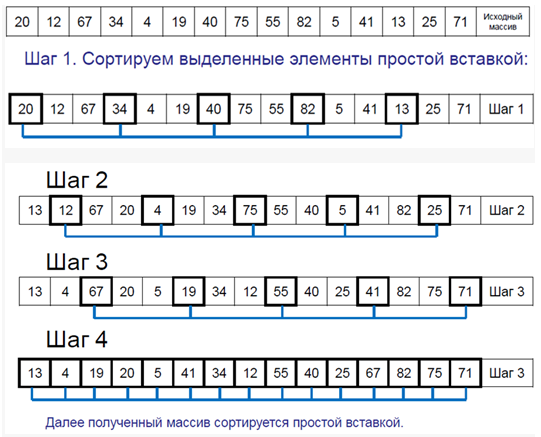


**Сортировка Шелла.**

Сортировка Шелла основана на сортировке вставками. Идея сортировки методом Шелла состоит в том, чтобы сортировать элементы отстоящие друг от друга на некотором расстоянии, которое обычно называют step. Затем сортировка повторяется при меньших значениях step, и в конце процесс сортировки Шелла завершается при step = 1 (а именно обычной сортировкой вставками).

В свою очередь, основная идея алгоритма вставками — разделение исходного массива на две части, отсортированную и неотсортированную. На каждом шаге алгоритма число переходит от неотсортированной к отсортированной части.

Стр.4



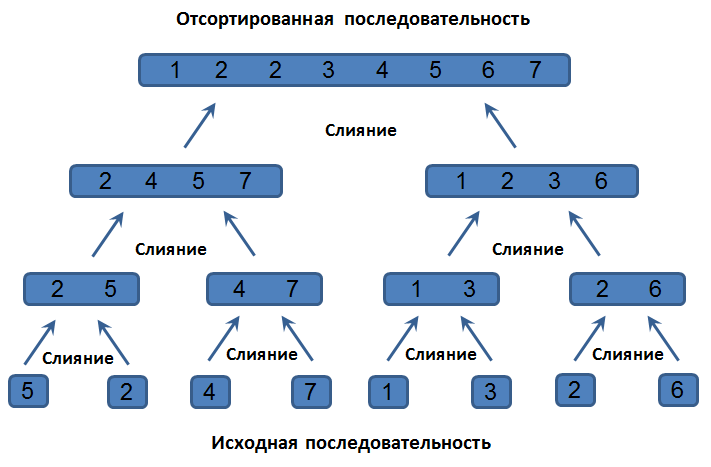
**Сортировка Слиянием.**

Эта сортировка основана на идее объединения двух отсортированных массивов.

Основная идея: Сортируемый массив разбивается на две части примерно одинакового размера. Каждая из получившихся частей сортируется отдельно 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 (так как любой массив длины 1 можно считать упорядоченным). Затем два упорядоченных массива соединяются в один. 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

Стр.5

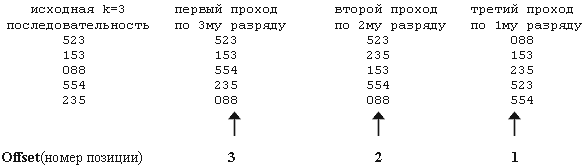
все оставшиеся элементы второго подмассива в результирующий массив



**Поразрядная сортировка.**

 Это один из алгоритмов, которые не используют сравнение элементов массива.

Основная идея: Сортировать по своему младшему(последнему) разряду, затем следующему(предпоследнему) и т.д. до старшего разряда, первого.   
До сортировки необходимо знать два параметра: k и d, где:  
k - количество разрядов в самом большом числе.  
m - разрядность данных: количество возможных значений разряда числа. С целых беззнаковых чисел эта идея переносится и на знаковые, с плавающей точкой. Число рассматривается в байтовой форме и в качестве разряда берется 1 байт. Таким образом для чисел определенного типа k и m становятся постоянными.

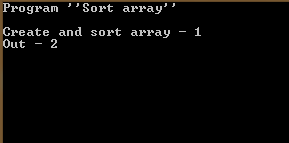


Стр.6

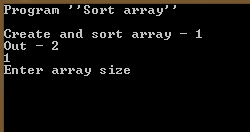
**Руководство пользователя**

1. При запуске программы пользователю предлагается создать и отсортировать массив, или завершить работу программы. При вводе «2» программа завершит

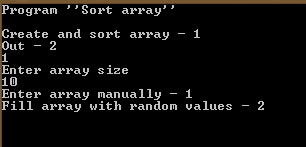
работу.



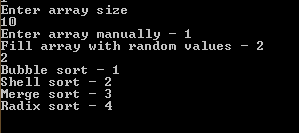
1. При вводе «1» программа предложит ввести размер массива:



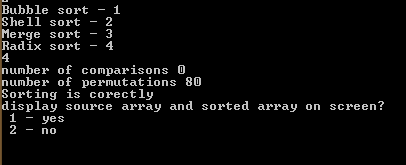
1. После ввода размера массива пользователю предлагается выбрать способ его заполнения. «1» - Заполнить массив вручную и «2» - Заполнить массив случайными числами:

 Стр.7

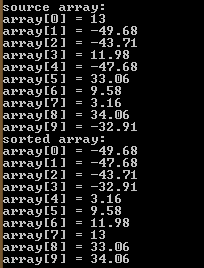
1. Затем показываются возможные способы сортировки (Пузырьком, Шелла, Слиянием и Поразрядной). Каждому методу сортировки соответствует ввод цифры, стоящей напротив.



1. После выбора метода сортировки, массив сортируется. На экран выводятся количество сравнений, перестановок, информация о том, успешно ли отсортирован массив, а так же спрашивается, нужно ли вывести на экран исходный массив, и отсортированный.



1. При вводе «1» массивы выводятся, и программа завершает работу, при вводе «2» программа зевершает работу сразу.

 Стр.8

# Описание программной реализации

Программа состоит из Lab1.cpp файла, и из bubblesort.h shellsort.h radix.h заголовочных файлов (для удобства подсчета количества сравнений и перестановок mergeSort пришлось перенести в основной файл).

Основными функциями программы являются:

1. void bubblesort() - функция, реализующая сортировку пузырьком.

Принимает указатель на исходный массив, и его размер. Ничего не возвращает.

1. void shellsort() - функция, реализующая сортировку Шелла. Принимает указатель на исходный массив, и его размер. Ничего не возвращает.
2. void mergeSort() - функция, реализующая сортировку слиянием. Принимает указатель на исходный массив, крайний левый и крайний правый индексы массива. Далее рекуррентно вызывает саму себя, разбивая исходный массив на подмассивы и сортируя их тем же способом.
3. void radix\_sort()- функция, реализующая поразрядную сортировку. Принимает указатель на исходный массив, и его размер. Вызывает внутри себя вспомогательные функции, описанные далее.

Вспомогательные функции:

1. void create\_counters() - создает массив счетчиков, для поразрядной сортировки. Принимает указатель на исходный массив, указатель на массив, выделенный под счетчики сортировки и размер исходного массива.
2. void radix\_pass() - создает массив смещений элементов исходного массива для поразрядной сортировки. Принимает номер байта, размер массива, указатель на исходный массив, указатель на массив для записи отсортированных по текущему байту элементов исходного массива, указатель на текущий участок массива counters.
3. void radix\_last\_pass() - корректирует положение отрицательных чисел, работая со старшими байтами для поразрядной сортировки. Принимает номер байта, размер массива, указатель на исходный массив, указатель на массив для записи отсортированных по текущему байту элементов исходного массива, указатель на текущий участок массива counters.

Стр. 9

# Подтверждение корректности

Для подтверждения корректности в программе реализована функция

void sorting\_check(), принимающая указатель на уже отсортированный массив, и его размер. В функции сравнивается предыдущий и следующий элемент массива, начиная со второго и заканчивая последним. В случае хотя бы одного элемента, стоящего не по порядку на экран выводится «Sorting is not correctly». В обратном случае на экран выводится «Sorting is correctly».

Стр.10

# Результаты экспериментов

Таблица сравнений:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 100 | | | 200 | | | | 400 | | | 600 | | | 1000 | | |
|  | Х | С | Л | | | Х | С | Л | | | | Х | С | Л | | | Х | С | Л | | | Х | С | Л | | |
| Пузырьком | 4950 | 4929 | 99 | 19900 | 19899 | 199 | | 79800 | 79734 | 399 | 179700 | 178619 | 599 | 499500 | 497904 | 999 |
| Шелла | ------- | 899 | 503 | ------- | 950 | 1203 | | -------- | 5494 | 2803 | ---------- | 8759 | 4804 | --------- | 16258 | 8006 |
| Слиянием | ----- | 300 | 356 | ------ | 651 | | 812 | ------ | 1505 | 1824 | ------- | 2419 | 2916 | -------- | 4381 | 5044 |
| Поразрядная | ----- | ------ | ---- | ------ | ------ | | ---- | ------ | ------- | ---- | ------- | -------- | ---- | -------- | ------ | ------ |

Таблица перестановок

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 100 | | | 200 | | | | 400 | | | 600 | | | 1000 | | |
|  | Х | С | Л | | | Х | С | Л | | | | Х | С | Л | | | Х | С | Л | | | Х | С | Л | | |
| Пузырьком | 4892 | 2371 | 0 | 19793 | 9779 | 0 | | 79600 | 38475 | 0 | 179402 | 89873 | 0 | 499003 | 236031 | 0 |
| Шелла | ------- | 354 | 43 | ------- | 989 | 44 | | ------- | 2608 | 45 | ---------- | 3668 | 46 | --------- | 8022 | 46 |
| Слиянием | ------ | 672 | 672 | ------ | 1544 | | 1544 | ------- | 3488 | 3488 | --------- | 5576 | 5576 | -------- | 9976 | 9976 |
| Поразрядная | ------ | 402 | ---- | ------ | 1600 | | ---- | ------- | 3200 | ---- | -------- | 4800 | ---- | -------- | 8000 | ------ |

Х – Худший случай (Обратно отсортированный массив)  
С – Средний случай (Случайный массив)  
Л – Лучший случай (Отсортированный массив)

1. Сортировка пузырьком

**Сложность алгоритма**:

O(n) в лучшем случае.

O(n^2) в среднем.

O(n^2) худшем случае.

Видно, что число сравнений действительно растет линейно в лучшем случае. В худшем и среднем сложность сравнений и перестановок растет квадратично.

1. Сортировка Шелла

**Сложность алгоритма**:

O(n) в лучшем случае. Стр.11

O(n (log n)²) в среднем случае.

Сложность худшего случая равна сложности в среднем случае. Она зависит не от исходных данных, а от выбора шага step, и потому не рассматривается.

По таблице видно, что в лучшем случае сложность меняется линейно, так как число перестановок постоянно.

1. Сортировка слиянием

**Сложность алгоритма**:

O(n\*log(n)) в лучшем случае.

O(n\*log(n)) в среднем случае .

Сложность в худшем случае здесь так же равна сложности в среднем, и этот случай так же не просто получить, поэтому он не рассматривается.

По таблице хорошо видно, что сложность и в лучшем и худшем случае меняется одинаково линейно-логарифмически.

1. Сортировка слиянием

**Сложность алгоритма**:

Сложность поразрядной сортировки всегда линейно зависит от n числа элементов и равна O(n\*(m+k)), где m и k постоянные для каждого отдельного случая. Значит достаточно рассмотреть, например, средний случай. Легко убедится, что сложность линейная.

Стр.12

# Заключение

С помощью написанной программы удалось эмпирически доказать сложность работы 4 алгоритмов сортировки. Основываясь на этих данных можно понять для какого случая какая сортировка лучше подходит. Например сортировка пузырьком подходит для почти отсортированных массивов, а сортировка слиянием для случаев, когда неизвестно заранее какие данные в массиве, ведь не зависимо от этого ее сложность O(n\*log(n)), и вероятность выпадения плохого случая мала .

Стр.13

# Приложение

**Сортировка пузырьком.**

void bubblesort(double\* mas, int size)

{

int count\_comp = 0, count\_perm = 0;

int flag;

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

{

flag = 0;

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

{

count\_comp++;

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

{

count\_perm++;

double swap = mas[j];

mas[j] = mas[j + 1];

mas[j + 1] = swap;

flag = 1;

}

}

if (flag == 0)

break;

}

std::cout << "number of comparisons " << count\_comp << std::endl;

std::cout << "number of permutations " << count\_perm << std::endl;

}

**Сортировка Шелла.**

void shellsort(double\* mas, int size)

{

int count\_comp = 0, count\_perm = 0;

double swap;

int i, j, k, step;

for (step = size / 2; step > 0; step /= 2)

{

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

{

for (j = i + step; j < size; j += step)

{

swap = mas[j];

k = j - step;

count\_comp++;

while (k >= 0 && mas[k] >= mas[k + step])

{

count\_comp++;

count\_perm++;

mas[k + step] = mas[k];

mas[k] = swap;

k = k - step;

}

}

}

}

std::cout << "number of comparisons " << count\_comp << std::endl;

std::cout << "number of permutations " << count\_perm << std::endl;

} Стр.14

**Сортировка слиянием.**

//переменные count\_copm1 count\_perm1 здесь глобальные

void mergeSort(double\* mas, int left\_half, int right\_half)

{

if (left\_half == right\_half) return;

int mid = (left\_half + right\_half) / 2;

mergeSort(mas, left\_half, mid);

mergeSort(mas, mid + 1, right\_half);

int i = left\_half;

int j = mid + 1;

double\* tmp = (double\*)malloc((right\_half - left\_half + 1) \* sizeof(double));

for (int step = 0; step < right\_half - left\_half + 1; step++)

{

count\_comp1++;

if ((j > right\_half) || ((i <= mid) && (mas[i] < mas[j]) ))

{

count\_perm1++;

tmp[step] = mas[i];

i++;

}

else

{

count\_perm1++;

tmp[step] = mas[j];

j++;

}

}

for (int step = 0; step < right\_half - left\_half + 1; step++)

{

mas[left\_half + step] = tmp[step];

}

free(tmp);

}

**Поразрядная сортировка.**

int count\_comp = 0, count\_perm = 0;

void create\_counters(double\* mas, long\* counters, long size)

{

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

unsigned char\* flag = (unsigned char\*)mas;

unsigned char\* end\_of\_mas = (unsigned char\*)(mas + size);

while (flag != end\_of\_mas)

{

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

counters[256 \* i + \*flag++]++;

}

}

void radix\_pass(short Offset, long size, double\* mas, double\* dest, long\* count)

{

long temp\_sum = 0, \* count\_flag = count, buf;

for (int i = 256; i > 0; --i, ++count\_flag) Стр.15

{

buf = \*count\_flag;

\*count\_flag = temp\_sum;

temp\_sum += buf;

}

unsigned char\* flag = (unsigned char\*)mas + Offset;

double\* mas\_flag = mas;

for (long i = size; i > 0; --i, flag += sizeof(double), ++mas\_flag)

{

count\_perm++;

count\_flag = count + \*flag;

dest[\*count\_flag] = \*mas\_flag;

++(\*count\_flag);

}

}

void radix\_last\_pass(short Offset, long size, double\* mas, double\* dest, long\* count)

{

long buf, \* count\_flag = count;

long num\_neg = 0;

for (int i = 128; i < 256; i++)

num\_neg += count[i];

long temp\_sum = num\_neg;

for (int i = 0; i < 128; ++i, ++count\_flag)

{

buf = \*count\_flag;

\*count\_flag = temp\_sum;

temp\_sum += buf;

}

temp\_sum = count[255] = 0;

count\_flag = count + 254;

for (int i = 254; i >= 128; --i, --count\_flag)

{

\*count\_flag += temp\_sum;

temp\_sum = \*count\_flag;

}

unsigned char\* flag = (unsigned char\*)mas + Offset;;

double\* mas\_flag = mas;

for (long i = size; i > 0; --i, flag += sizeof(double), ++mas\_flag)

{

count\_perm++;

count\_flag = count + \*flag;

if (\*flag < 128) dest[(\*count\_flag)++] = \*mas\_flag;

else dest[--(\*count\_flag)] = \*mas\_flag;

}

}

void radix\_sort(double\*& source\_mas, long size)

{

double\* out\_mas = (double\*)malloc(size \* sizeof(double));

unsigned short i;

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

create\_counters(source\_mas, counters, size); Стр.16

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

{

count = counters + 256 \* i;

radix\_pass(i, size, source\_mas, out\_mas, count);

double\* ttemp;

ttemp = source\_mas;

source\_mas = out\_mas;

out\_mas = ttemp;

}

count = counters + 256 \* i;

radix\_last\_pass(i, size, source\_mas, out\_mas, count);

delete source\_mas;

source\_mas = out\_mas;

free(counters);

std::cout << "number of comparisons " << count\_comp << std::endl;

std::cout << "number of permutations " << count\_perm << std::endl;

}

Стр.17