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 3823Б1ПМ1

Калинин Б. А.

**Проверил**:

преподаватель каф. ВВСП,

Волокитин В.Д.

Нижний Новгород

2023

**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

В данной лабораторной работе была поставлена задача реализовать на языке C 4 сортировочных алгоритма для ТД float:

* Сортировка пузырьком (с выходом)
* Сортировка расчёской
* Сортировка слиянием
* Поразрядная сортировка

Также была поставлена задача реализовать проверку на правильность сортировочных алгоритмов, замеры времени их работы и показать сложность каждого алгоритма.

# Метод решения

**Сортировка пузырьком** проходит по списку чисел многократно, сравнивая каждую пару соседних элементов и меняя их местами, если они находятся в неправильном порядке. Основная идея заключается в том, что большие элементы всплывают (поднимаются) к концу списка, как пузырьки в воде, отсюда и название "сортировка пузырьком". В итоге, после каждого прохода самый большой элемент "всплывает" на правильную позицию. Алгоритм повторяет эти проходы до тех пор, пока весь массив не будет отсортирован. [1]

**Сортировка расчёской** - это усовершенствованный вариант сортировки пузырьком, который позволяет ускорить процесс сортировки. Она получила своё название из-за визуальной аналогии с расчёской волос. Основная идея сортировки расчёской заключается в том, чтобы "разреживать" элементы списка с помощью большого шага и затем постепенно уменьшать этот шаг, пока он не станет равным 1. Это уменьшение шага уменьшает количество перестановок элементов и повышает эффективность сортировки.

**Сортировка слиянием** работает за счёт слияния 2 отдельных отсортированных частей массива в одну отсортированную часть, таким образом алгоритм предполагает сначала объединения в пары каждых 2 соседних элементов (подмассив из 1 элемента отсортирован), затем в четвёрки каждых 2 соседних пар, и так далее каждый раз увеличивая шаг разбиения массива на подмассивы в 2 раза, пока он не станет больше размера самого массива, к этому моменту массив будет отсортирован.

**Поразрядная сортировка** предполагает разбиение всех элементов массива на разряды, по которым будет происходит сортировка. В данной лабораторной работе реализован вариант LSD (least significant digit), в котором сортирование происходит от младшего разряда к старшему. При сортировке каждого разряда допустимо применение сортировки подсчётом так как число различных значений разряда нам заранее известно и невелико.

# Руководство пользователя

При запуске программы пользователю будет предложено начать работу с ней или выйти. Для начала работы следует ввести 1, для выхода – 2.

Далее (при начале работы) пользователю будет предложено выбрать способ, которым хочет задать массив – вручную или случайно (значения будут в диапазоне [-1000; 1000])

После выбора пользователь должен ввести длину массива N, и в случае задания вручную после пользователь должен ввести N чисел.

Далее на экран будет выведено меню сортировок, из которого пользователь должен выбрать ту сортировочную функцию которую он хочет проверить.

После выбора на экран будет выведено количество тактов в течении которых работала функция сортировки, и будет выведен результат проверки правильности сортировки.

# Описание программной реализации

Проект представлен одним файлом, содержащим все поставленные к реализации виды сортировок, проверку на правильность, время и пользовательское меню.

**Проверка на правильность** была реализована в функции check\_sorted\_float(float\* sorted, float\* orig, int n), которая берёт в качестве аргументов указатели на отсортированный другой функцией массив, его оригинал, и длину. Сама проверка происходит за счёт сортировки оригинала с помощью функции стандартной библиотеки qsort, и последующей проверки на совпадение отсортированного оригинала с требующим проверки отсортированным массивом.

**Замеры времени** были реализованы в функции clock\_measure(float\* mas, int n, void(\*sort)(float\*, int)), которая берёт в качестве аргумента неотсортированный массив, его длину и указатель на функцию сортировки. При помощи вызова \_\_rdtsc() замеряется количество тактов прошедших с момента последнего сброса счётчика перед началом сортировки, вызывается функция sort для сортировки массива, после чего снова замеряется кол-во пройденных тактов. Время работы будет разностью значений, возвращаемых \_\_rdtsc() до и после работы сортировки и будет представлено как количество тактов, которые она работала.[2]

Все реализованные функции сортировок берут в качестве аргументов указатель на массив и его длину, сортируют его и ничего не возвращают.

**Сортировка пузырьком** представлена в программе функцией bubble\_sort\_float(float\* mas, int size), Внешний цикл for (i = 0; i < size - 1; i++) проходит по всем элементам массива, а внутренний цикл for (j = 0; j < size - i - 1; j++) выполняет сравнения и обмен элементов. Выход реализован с помощью флажка f, если после работы внутреннего цикла f = 1, обмена элементов не произошло, массив отсортирован и внешний цикл прекращает работу.

**Сортировка расчёской** представлена в программе функцией comb\_sort\_float(float\* mas, int size), Внешний цикл while (step >= 1) выполняет проходы по массиву с уменьшением шага сортировки, а внутренний цикл for (j = 0; j < n - step; j++) выполняет сравнения и обмен элементов. Значение переменной factor вычисляется с использованием формулы для оптимального коэффициента уменьшения шага в алгоритме сортировки расческой.

**Сортировка слиянием** представлена в программе функцией merege\_sort\_float(float\* mas, int size) и вспомогательной merge(float\* a, int an, float\* b, int bn, float\* res)

merge\_sort\_float вначале выделяет память для временного массива "mas2" и инициализирует переменные "swap\_count" и "step". Затем происходит несколько итераций, на каждой из которых происходит объединение отсортированных пар массивов с постепенно увеличивающимися размерами, а затем меняются указатели местами. Этот процесс повторяется до тех пор, пока размер объединяемых пар не станет равным размеру исходного массива. После завершения этой фазы, проверяется нечетность количества слияний, и если нечётное - указатели массивов меняются местами.

merge принимает два отсортированных массива a и b размерами an и bn соответственно, и объединяет их в один отсортированный массив res. Она использует индексы ai и bi для отслеживания текущих позиций в массивах a и b соответственно, а также индекс k для отслеживания текущей позиции в результирующем массиве res. Происходит сравнение элементов массивов a и b, и меньший элемент помещается в результирующий массив, затем указатели сдвигаются вправо. Этот процесс продолжается, пока один из указателей не дойдет до конца своего массива. Затем оставшиеся элементы просто добавляются в результирующий массив.

**Поразрядная сортировка** представлена в программе функцией radix\_sort\_float(float\* mas, int size) и двумя необходимыми для её работы radix\_sort\_uint(unsigned int\* mas, int size) и offset(int count[256], unsigned char\* mas, int size, int offset)

radix\_sort\_float создаёт новый указатель ptr на элементы массива как на unsigned int, далее с элементы массива преобразуются путём побитовых операций таким образом: отрицательные числа преобразуются в обратный код, а в отрицательные числа в старший бит записывается 1. После этого по указателю на uint вызывается сортировка radix\_sort\_uint, и выполняются обратные битовые преобразования.

radix\_sort\_uint создаёт вспомогательный указатель на pm, выделяет память под вспомогательный второй массив mas2, создаёт массив count[256], далее в цикле по каждому байту (от 0 до sizeof(unsigned int)-1): вызывает offset, чтобы собрать список начал по данному разряду, во внутреннем цикле переписывает элементы из исходного массива во второй массив, используя индексы из массива count (сортируя их по текущему разряду), меняет местами указатели на исходный и вспомогательный массивы и обновляет pm как указатель на новый текущий массив. После сортировки освобождает память выделенную вспомогательный массив.

offset работает с массивом count, сначала обнуляет его, подсчитывает все встречающиеся значения текущего разряда offset, проходя по массиву байтов mas с шагом sizeof(uint) и отступом offset, далее преобразует массив count в массив индексов начал элементов по разряду, используя подсчет префиксной суммы и сдвиг вправо.

# Подтверждение корректности

Для подтверждения корректности выполняемых сортировок в программе используется сортировочная функция стандартной библиотеки языка С qsort, представляющая из себя модифицированный вариант сортировки Хоара. Для проверки исходный массив сортируется сначала проверяемой сортировкой, затем стандартной, и полученные отсортированные массивы сравниваются между собой на полное совпадение. Если какой-либо элемент в двух отсортированных массивах не совпадает, то функция сортировки работает некорректно, иначе – корректно.

# Результаты экспериментов

В результате экспериментов были получены данные для того чтобы наглядно показать сложность работы каждого реализованного сортировочного алгоритма. Для того чтобы показать это используется график зависимости отношения проработанного времени(в тактах) к предполагаемой сложности алгоритма (t/O(n)) от размера массива n.

Далее приведены составленные графики:

Рисунок

Рисунок

Рисунок

Рисунок

По полученным зависимостям видно, что достаточно больших значениях n данные отношения стремятся к константе, т. е. время работы отличаться от сложности лишь множителем, что свидетельствует о том что предполагаемая сложность алгоритма является правильной.

# Заключение

В лабораторной работе были реализованы все необходимые сортировочные алгоритмы и инструменты для оценки времени и корректности их работы, была наглядно показана сложность каждого алгоритма.

По результатам работы можно сделать следующие выводы:

* Сортировка пузырьком является приемлемой только при малых значениях n иначе, её время работы будет
* Сортировка расчёской имеет лучшую сложность чем у пузырька, однако требует достаточно точного вычисления коэффициента уменьшения шага, иначе она будет работать некорректно.
* Сортировка слиянием в данной реализации работает быстрее чем расчёска на больших n, но требует выделения дополнительной памяти.
* Поразрядная сортировка работает лучше всех остальных на больших n, но требует выделения дополнительной памяти и реализации под каждый отдельный тип данных.

Таким образом все сортировочные алгоритмы имеют своё применение в решении различных проблем и даже могут использоваться совместно (например как qsort в С)

# Источники

[1] Сортировка пузырьком — Википедия (<https://ru.wikipedia.org/wiki/Сортировка_пузырьком>)

[2] \_\_rdtsc | Microsoft Learn (<https://learn.microsoft.com/ru-ru/cpp/intrinsics/rdtsc?view=msvc-170>)

# Приложение

// ПРОВЕРКА

int compare\_func(const void\* x1, const void\* x2) {

if (\*(float\*)x1 < \*(float\*)x2) {

return -1;

}

if (\*(float\*)x1 == \*(float\*)x2) {

return 0;

}

return 1;

}

int check\_sorted\_float(float\* sorted, float\* orig, int n) {

int i;

qsort(orig, n, sizeof(float), compare\_func);

for (i = 0; i < n; i++) {

if (orig[i] != sorted[i]) {

return 0;

}

}

return 1;

}

// ПУЗЫРЁК

void bubble\_sort\_float(float\* mas, int size) {

int i, j, f;

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

f = 1;

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

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

swap\_float(&mas[j], &mas[j + 1]);

f = 0;

}

}

if (f) {

break;

}

}

}

// РАСЧЁСКА

void comb\_sort\_float(float\* mas, int size) {

int step = size - 1;

double factor = 1.0 / (1.0 - exp(-(1 + sqrt(5)) / 2.0));

int j;

while (step >= 1) {

for (j = 0; j < size - step; j++) {

if (mas[j] > mas[j + step]) {

swap\_float(&mas[j], &mas[j + step]);

}

}

step /= factor;

}

}

// СЛИЯНИЕ

void merge(float\* a, int an, float\* b, int bn, float\* res) {

int ai, bi, k;

k = 0;

ai = 0;

bi = 0;

while (ai < an && bi < bn) {

if(a[ai] < b[bi]) {

res[k++] = a[ai++];

}

else {

res[k++] = b[bi++];

}

}

for (; ai < an; ai++) {

res[k++] = a[ai];

}

for (; bi < bn; bi++) {

res[k++] = b[bi];

}

return;

}

void merge\_sort\_float(float\* mas, int size) {

float\* mas2;

int i;

int swap\_count = 0;

int step = 1;

mas2 = (float\*)malloc(size \* sizeof(float));

for (; step < size; step \*= 2) {

for (i = 0; i < size; i += 2 \* step)

merge(&mas[i], max(0, min(step, size - i)), &mas[i + step], max(0, min(step, size - step - i)), &mas2[i]);

swap\_pointers\_float(&mas, &mas2);

swap\_count++;

}

if (swap\_count % 2) {

swap\_pointers\_float(&mas, &mas2);

}

free(mas2);

}

// ПОРАЗРЯДНАЯ

void offset(int count[256], unsigned char\* mas, int size, int offset) {

int i;

null\_array\_int(count, 256);

//подсчёт кол-ва элементов по значениям данного разряда

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

count[mas[offset + i\*sizeof(unsigned int)]]++;

}

//перевод count в массив индексов начал элементов по разряду (подсчёт префиксной суммы и её сдвиг вправо)

for (i = 1; i < 256; i++) {

count[i] += count[i - 1];

}

for (i = 255; i>0; i--) {

count[i] = count[i - 1];

}

count[0] = 0;

}

void radix\_sort\_uint(unsigned int\* mas, int size){

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

unsigned int\* mas2 = (unsigned int\*)malloc(size \* sizeof(unsigned int));

int count[256];

int i, j;

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

offset(count, pm, size, i); //собирает список начал по байту i

for (j = 0; j < size; j++) {

mas2[count[pm[j \* sizeof(unsigned int) + i]]++] = mas[j];

//записывает текущий элемент первого массива во второй

//при этом выбирает для записи такой индекс, который соответствует текущему номеру для записи чисел с данным i-м разрядом (из count)

//увеличивает count[значение i-го байта], для того, чтобы следующий элемент таким же с i-м байтом был записан в последующую ячейку

}

swap\_pointers\_uint(&mas, &mas2);

pm = (unsigned char\*)mas; // обновление pm

}

free(mas2); //так-как произошло чётное количество swap'ов (sizeof(uint)/sizeof(char) = 4) исходный массив остался в mas

}

void radix\_sort\_float(float\* mas, int size){

int i;

//реализация для флотов происходит путём преобразования (float в uint) с сохранением их порядка

//основана на том, что все биты в реализации float'а распологаются в порядке убывания "важности"

unsigned int\* ptr;

ptr = (unsigned int\*)mas;

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

if (ptr[i] >> 31) {

//отрицательные числа преобразуются в обратный код

//обратный код позволяет инвертировать порядок отрицательных чисел (чем больше модуль отрицательного числа тем оно меньше)

//также меняет самый старший бит на 0 делая их меньше всех положительных

ptr[i] = (ptr[i] ^ 4294967295) + 1;

}

else {

//неотрицательные числа изначально расположены в правильном порядке после преобразования в uint

//запись 1 в старший бит позволяет расположить неотрицательные числа после отрицательных;

ptr[i] = ptr[i] ^ 2147483648;

}

}

radix\_sort\_uint(ptr, size);

//обратное преобразование (из uint в float)

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

if (ptr[i] >> 31) {

ptr[i] = ptr[i] ^ 2147483648;

}

else {

ptr[i] = (ptr[i] - 1) ^ 4294967295;

}

}

}

// ЗАМЕРЫ ВРЕМЕНИ

unsigned \_\_int64 clock\_measure(float\* mas, int n, void(\*sort)(float\*, int)) {

unsigned \_\_int64 start, end;

start = \_\_rdtsc();

sort(mas, n);

end = \_\_rdtsc();

return end - start;

}