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

**Проверил:**

преподаватель каф. ВВСП,

Волокитин В.Д.

Нижний Новгород

2023

**Содержание**

[Постановка задачи 4](#_Toc154702149)

[Метод решения 5](#_Toc154702150)

[Руководство пользователя 7](#_Toc154702151)

[Описание программной реализации 8](#_Toc154702152)

[Подтверждение корректности 9](#_Toc154702153)

[Результаты эксперимента 10](#_Toc154702154)

[Заключение 13](#_Toc154702155)

[Приложение 14](#_Toc154702156)

[Источники 17](#_Toc154702157)

**Постановка задачи**

Целью лабораторной работы является изучение и анализ эффективности различных алгоритмов сортировки на практике. Задачей исследования является реализация и сравнение четырех различных методов сортировки, а именно: сортировка вставками (insertion sort), сортировка Хоара (quick sort), сортировка слиянием (merge sort) и поразрядная сортировка (radix sort). В ходе работы будет проведено сравнение времени выполнения каждого из алгоритмов на различных объемах данных, что позволит оценить их производительность и применимость в различных сценариях использования.

Кроме того, целью исследования является выявление особенностей каждого из алгоритмов сортировки, их преимуществ и недостатков. Такой подход позволит определить оптимальный выбор метода сортировки в зависимости от конкретной задачи и объема данных.

**Метод решения**

**1.Сортировка вставками** (Insertion sort). Будем считать, что массив поделен на отсортированную левую и неотсортированную правую часть. В начале работы алгоритма в левую часть помещается первый элемент исходного массива, подмассив из одного элемента считается отсортированным. Каждый следующий цикл алгоритм берет первый элемент из правой части (ключ), помещает его в левую и с помощью сравнений и перестановок (i и i-1 элемента) ключ достигает правильного места в отсортированной части. Если на вход алгоритму поступил отсортированный массив, то он проверит его отсортированность за n операций, где n – мощность массива, и завершит работу. В лучшем случае временная сложность алгоритма – O(n), в худшем и среднем случае – O(n2) (ист. 1).

**2.Сортировка слиянием** (Merge sort). Алгоритм разбивает исходный массив из n элементов на n отсортированных массивов длины 1, затем попарно применяет процедуру слияния массивов до тех пор, пока не останется один отсортированный массив длины n. Эта процедура заключается в том, что мы сравниваем элементы массивов (начиная с начала) и меньший из них записываем в финальный. И затем, в массиве у которого оказался меньший элемент, переходим к следующему элементу и сравниваем теперь его. В конце, если один из массивов закончился, мы просто дописываем в финальный другой массив. После мы наш финальный массив записываем вместо двух исходных и получаем отсортированный участок (ист. 2). Алгоритм слияния имеет временную сложность O(n). Сортировка слиянием в любом случае имеет сложность O(n\*log(n)).

**3.Сортировка Хоара** (quick sort).Общая идея алгоритма состоим в следующем: сначала выбирается элемент из массива, называемый опорным. Если в качестве опорного выбрать первый элемент фрагмента массива или последний, то если массив уже упорядочен, сложность алгоритма будет O(n2), так как на каждом рекурсивном вызове от большей части массива будет отделяться всего один элемент (ист. 3). Поэтому в алгоритме быстрой сортировки Хоара, как правило, в качестве опорного элемента выбирается случайный элемент массива. Сравниваем все остальные элементы с опорным и переставляем их в массиве так, чтобы разбить массив на два непрерывных отрезка, следующих друг за другом: «элементы меньшие опорного» и «большие или равны». Для этих отрезков значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы. Временная сложность алгоритма: в лучшем и среднем случае – O(n\*log(n)). В худшем – O(n2) (ист. 4).

**4. Поразрядная сортировка** (Radix sort). Алгоритм пригоден для сортировки любых объектов, запись которых можно поделить на «разряды», содержащие сравнимые значения, например, строки, и вообще любые данные, представленные в виде набора байтов. Для поразрядной сортировки произвольных чисел используется их байтовое представление. Она основана на сортировке подсчетом. Алгоритм сортировки работает по следующему принципу: создается вспомогательный массив мощностью K+1, где K – разность максимального и минимального значения. Также необходимо прибавить 1 к ячейке вспомогательного массива с индексом, равным значению текущего элемента, ради возможного знака элемента. Тогда мощность вспомогательного массива будет иметь размер R = 256 – диапазон значений однобайтового беззнакового целого числа плюс единица. Сортировать числа по байтам будет сортировкой подсчетом. Сравнение 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 (ист.5). Сложность данного алгоритма в любом случае составляет O(V\*n), где V – объем исходного типа данных в байтах (ист.6).

**Руководство пользователя**

Взаимодействие программы и пользователя происходит через консоль. При вводе некорректных данных программа завершит работу. В ходе работы с программой пользователь может выбрать один из четырех типов сортировок, указать мощность массива. Массив автоматически заполняется случайными элементами, поэтому программа запрашивает дополнительные параметры: верхняя и нижняя граница множества элементов массива. По завершении работы программы пользователь может проверить корректность выполненной сортировки. Ниже приведен пример работы программы (рис. 1)

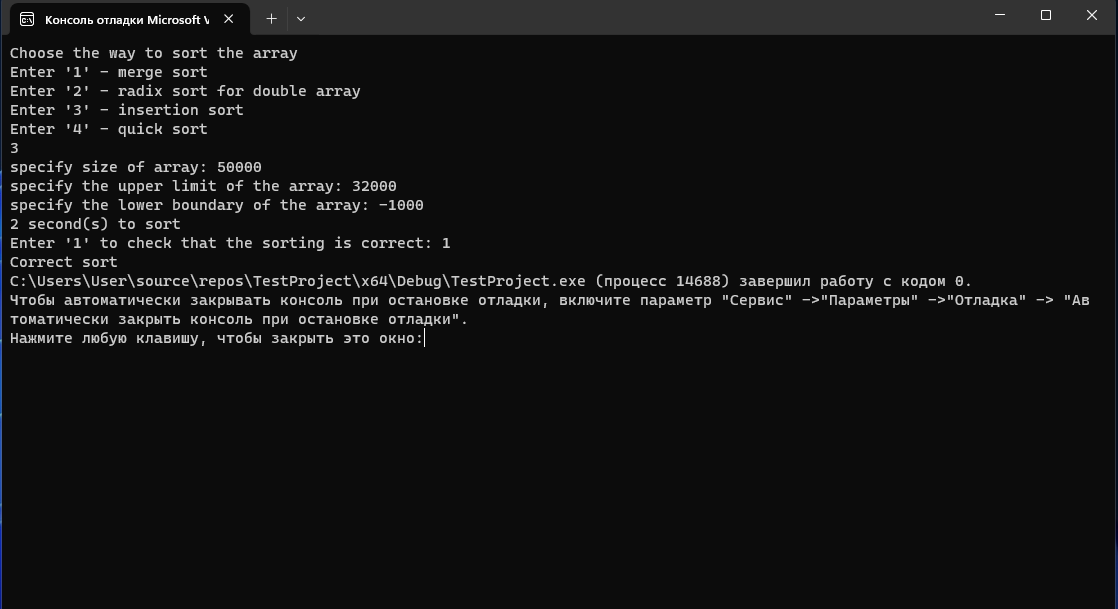


Рисунок 1. Пример работы с программой

**Описание программной реализации**

В папке проекта доступны следующие файлы:

1. SortLab.cpp. – файл реализации функций
2. SortLab.sln. – файл решения Visual Studio
3. SortLab.vcxproj. – файл проекта Visual Studio
4. Карпенко\_Руслан\_3823Б1ПМ1\_Отчет лабораторная.docx. – файл с отчетом по лабораторной работе

**Подтверждение корректности**

Для подтверждения корректности сортировки в программе реализована функция (рис.2), которая последовательно проверяет текущий и последующий элемент отсортированного массива и возвращает текстовый вывод о правильности работы алгоритма. Работа алгоритмов проверялась с массивом случайно сгенерированных чисел с дополнительным указанием верхней и нижней границы (рис.3).

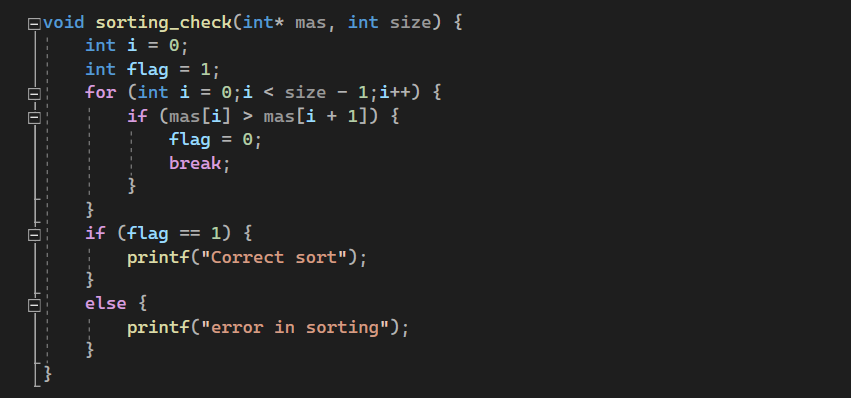


Рисунок 2. Функция, проверяющая корректность сортировки

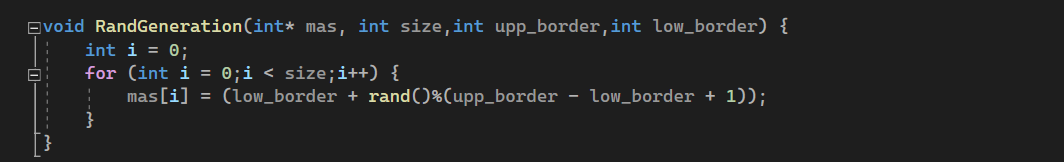


Рисунок 3. Функция для генерации случайных чисел в массиве

**Результаты эксперимента**

Приведем пример работы алгоритмов с разными объемами данных. Также рассмотрим график зависимости время выполнения сортировок от объема входных данных.

**Подтверждение сложности сортировок.**

**Cортировка вставками**. Предполагаемая сложность работы алгоритма O(n^2). На графике (рис. 4) заметно, что данная зависимость стремится к константе, поэтому теоретическая сложность верна

Рисунок 4. График отношения времени выполнения к предполагаемой сложности сортировки вставками.

**Сортировка Хоара.** Теоретическая заявленная сложность сортировки Хоара - O(n\*log(n)). Данная зависимость стремится к некоторой константе (рис. 5), поэтому заявленная сложность верна.

Рисунок 5. График отношения времени выполнения к предполагаемой сложности сортировки Хоара.

**Сортировка слиянием.** Предполагаемая сложность - O(n\*log(n)).

Предположение верно, потому что на графике (рис. 6) данная зависимость стремится к константе.

Рисунок 6. График отношения времени выполнения к предполагаемой сложности сортировки слиянием.

**Поразрядная сортировка.** Предполагаемая сложность - O(K\*n), где K – количество байт. Предположение верно, потому что на графике (рис. 7) зависимость стремится к константе.

Рисунок 7. График отношения времени выполнения к предполагаемой сложности сортировки поразрядная.

**Итог**: асимптотическая сложность совпала с предполагаемой. Алгоритмы сортировок на практике отработали схожим образом, как и предполагалось в теории.

**Заключение**

По результатам проведенной работы были сделаны следующие выводы:

1. Были реализованы и сравнены четыре различные методы сортировок, а именно: сортировка вставками (insertion sort), сортировка Хоара (quick sort), сортировка слиянием (merge sort) и поразрядная сортировка (radix sort). В ходе работы было проведено сравнение времени выполнения каждого из алгоритмов на различных объемах данных, что позволило оценить их производительность и применимость.
2. Практическая оценка сложности сортировки слиянием: O(n\*log(n)).
3. Сложность поразрядной сортировки: O(n).
4. Сложность сортировки вставками составила O(n2).
5. Сложность сортировки Хоара: O(n\*log(n)).

**Приложение**

void insertion(int\* mas, int size) {

int i, j;

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

j = i + 1;

while (j > 0 && mas[j - 1] > mas[j]) {

swap(&mas[j - 1], &mas[j]);

j--;

}

}

}

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

int ai = 0;

int bi = 0;

int di = 0;

while (ai < an && bi < bn) { //an,bn - размеры

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

data[di++] = a[ai++];

}

else {

data[di++] = b[bi++];

}

}

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

data[di++] = a[ai];

}

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

data[di++] = b[bi];

}

}

void mergesort(int\* m, int size) {

int counter = 0;

int step;

int i;

int\* tmpdata = (int\*)malloc(size \* sizeof(int));

for (step = 1; step < size; step \*= 2) {

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

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

}

swap\_p(&m, &tmpdata);

counter++;

}

if ((counter % 2) != 0) {

swap\_p(&m, &tmpdata);

}

free(tmpdata);

}

void foffset(unsigned char\* mas, int size, int offset, int count[257]) {

int i = 0;

int tmp;

for (i = 0; i < 257; count[i++] = 0) {};

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

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

}

tmp = count[0];

count[0] = 0;

for (i = 0; i < 256; i++) {

swap(&tmp, &count[i + 1]);

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

}

}

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

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

int count[257];

int i = 0, j = 0;

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

for (; i < sizeof(double); i++) {

foffset(pm, size, i, count);

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

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

}

double\* tempp;

tempp = mas;

mas = mas2;

mas2 = tempp;

pm = (unsigned char\*)mas;

}

free(mas2);

}

int partition(int\* mas,int s,int f) {

int a = mas[s + rand() % (f - s + 1)];

f--;

while (s < f) {

while (mas[s] < a) {

s++;

}

while (mas[f] > a) {

f--;

}

if (s < f) {

swap(&(mas[s]), (mas[f]));

s++;

f--;

}

}

return s;

}

void qsort(int s, int f, int\* mas) {

int p;

if ((f - s) < 2) {

return;

}

p = partition(mas, s, f);

qsort(s, p, mas);

qsort(p, f, mas);

return;

}

**Источники**

1. <https://ru.wikipedia.org/wiki/Сортировка_вставками>

2. <https://ru.wikipedia.org/wiki/Сортировка_слиянием>

3. [https://foxford.ru/wiki/informatika](https://foxford.ru/wiki/informatika/bystraya-sortirovka-hoara-python?utm_referrer=https%3A%2F%2Fyandex.ru%2F)

4. <https://ru.wikipedia.org/wiki/Быстрая_сортировка>

5. *Д. Кнут.* Искусство программирования, том 3.  «Вильямс», 2007. стр. 192 - 201. ISBN 5-8459-0082-4.

6. <https://thecode.media/radix/>