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 3822Б1-ПМ1

Вавилов В.В.

**Проверил**:

преподаватель каф. МОСТ,

Волокитин В.Д.

Нижний Новгород

2022

Оглавление

[Постановка задачи 3](#_Toc122681331)

[Метод решения 4](#_Toc122681332)

[Руководство пользователя 9](#_Toc122681333)

[Описание программной реализации 11](#_Toc122681334)

[Подтверждение корректности 14](#_Toc122681335)

[Результаты экспериментов 15](#_Toc122681336)

[Заключение 22](#_Toc122681337)

Приложение…………………………………………………………………………23

# Постановка задачи

Цель лабораторной работы - реализовать на языке программирования Си сортировку вставками, сортировку Шелла и поразрядную сортировку. Сортировки нужно осуществить для данных типа float. Программа должна работать корректно и быть удобной для пользователя, иметь понятный интерфейс. Пользователь должен иметь возможность выбора сортировки. Необходимо описать программную реализацию и алгоритмы работы данных сортировок. Нужно подтвердить корректность реализации данных сортировок. Провести эксперименты для подтверждения сложности, описать способ проведения экспериментов и сделать вывод по полученным результатам.

# Метод решения

Мною были реализованы три сортировки: сортировка вставками с линейным и бинарным поиском, сортировка Шелла и поразрядная сортировка.

1. **A) Сортировка вставками с линейным поиском**

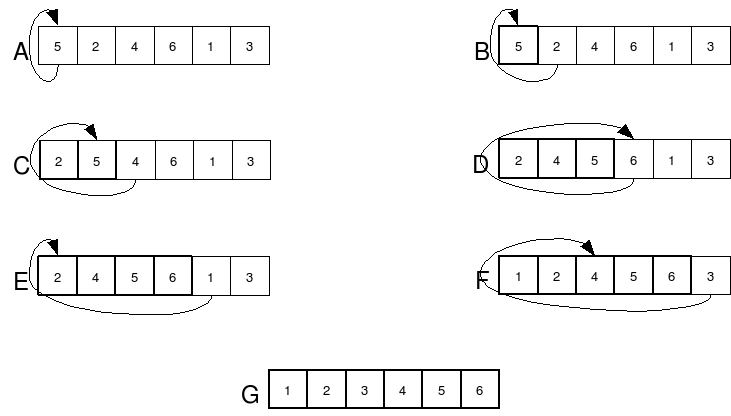


Рисунок 1 - иллюстрация работы сортировки вставками

Общая суть сортировок вставками: сначала перебираются элементы в неотсортированной части массива, а затем каждый элемент вставляется в отсортированную часть массива на то место, где он должен находиться. Главное преимущество сортировки вставками - очень быстрая обработка почти упорядоченных массивов.

То есть, из неотсортированной части извлекается элемент. Поскольку другая часть массива отсортирована, то в ней достаточно быстро можно найти своё место для этого извлечённого элемента. Элемент вставляется куда нужно, в результате чего отсортированная часть массива увеличивается, а неотсортированная уменьшается. В конечном счёте все элементы отсортируются и встанут на свои места.

Сам принцип работы алгоритма: проходим по массиву слева направо и обрабатываем по очереди каждый элемент. Слева от очередного элемента увеличиваем отсортированную часть массива, справа по ходу работы соритровки неотсортированная часть уменьшается. В отсортированной части массива ищется точка вставки для очередного элемента. Элементы, находящиеся справа от потенциальной точки вставки очередного элемента, сдвигаются на 1 единицу вправо, а выбранный элемент встаёт в нужное место, тем самым отсортированная часть массива увеличивается, а неотсортированная уменьшается. Спустя n-ое количество операций массив будет полностью отсортирован.

Сложность данного алгоритма сортировки равна O(n2) в худшем и среднем случае. А в лучшем случае O(n). В основном, лучшим случаем является ситуация, когда массив является почти упорядоченным . Ведь при близком расположении близких по величине данных место вставки почти всегда будет находиться близко к краю отсортированной части, что позволит вставлять элемент с наименьшими накладными расходами.

**B) Сортировка вставками с бинарным поиском**

Здесь для поиска места для вставки элемента a[k] в отсортированную часть массива используется алгоритм бинарного поиска. Элемент a[k] сравнивается, вначале с элементом [k/2]. Затем, если он меньше, поиск места осуществляется в первой половине упорядоченной части массива, т.е. элемент сравнивается с элементом [k/4], а если больше, поиск места осуществляется во второй половине упорядоченной части массива, т.е. элемент сравнивается с элементом [k/2] + [k/4| и т.д. до тех пор, пока для него не найдется место. То есть поиск сводится к тому, что вновь определяется значение серединного элемента в выбранной половине и сравнивается с элементом a[k]. Процесс продолжается до тех пор, пока не будет найден элемент со значением необходимого элемента или не станет пустым интервал для поиска. Все элементы массива, начиная с позиции вставки, сдвигаются на одну позицию вправо, освобождая место для k-го элемента.

Поиск места вставки не является критичным для временной сложности алгоритма, поэтому данная оптимизация здесь практически ни на что не повлияет.В случае почти отсортированного массива бинарный поиск может работать даже медленнее, так как он начинает с середины отсортированного участка, который, вероятно, будет находиться далеко от точки вставки. Поэтому сложность данного алгоритма сортировки такая же как для сортировки вставками с линейным поиском: O(n2) в худшем и среднем случае, в лучшем случае O(n).

1. **Сортировка Шелла**

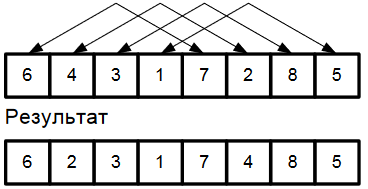


Рисунок 2 - иллюстрация работы сортировки Шелла (шаг 1)

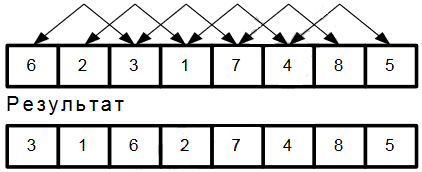


Рисунок 3 - иллюстрация работы сортировки Шелла (шаг 2)

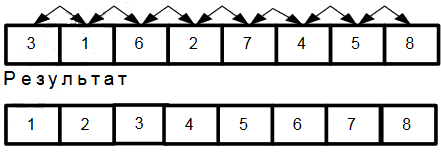


Рисунок 4 - иллюстрация работы сортировки Шелла (шаг 3)

Этот метод является улучшением сортировки вставками. При сортировке Шелла сначала сравниваются и сортируются между собой значения, стоящие один от другого на некотором расстоянии d. Алгоритм при первоначальных проходах достаточно быстро доводит массив к состоянию неполной упорядоченности, поскольку благодаря проходам с большим шагом, большинство малых по значению элементов перебрасываются в левую часть массива, а большинство крупных элементов массива попадают в правую. После этого процедура повторяется для некоторых меньших значений d d, а завершается сортировка Шелла упорядочиванием элементов при d = 1 d=1.Это означает, что на поздних стадиях сортировка сводится просто к перестановкам соседних элементов (то есть обычной сортировкой вставками).

В худшем случае сложность алгоритма составляет O(n2)  и O(n\* log2(n)) в лучшем случае. В среднем случае временная сложность алгоритма зависит от выбранных шагов.

1. **Поразрядная сортировка**

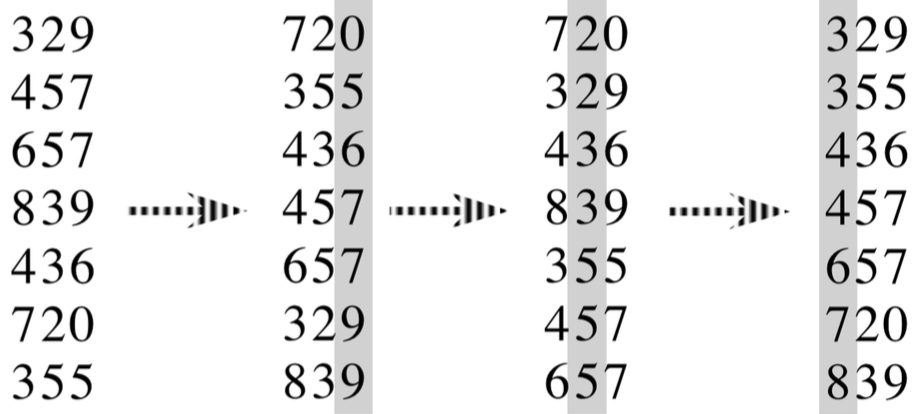
****

Рисунок 5 - иллюстрация работы поразрядной сортировки

Является расширенной версией сортировки подсчетом. 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 упорядочиваются по результатам сравнения значений этого разряда. Затем аналогично делается для следующего разряда, и так до конца, пока массив не будет полностью отсортирован.

# Руководство пользователя

После запуска программы пользователь может ввести количество элементов в массиве при нажатии на цифру 0, выбрать сортировку при нажатии на цифру 1 или выйти из программы при нажатии на цифру 2.

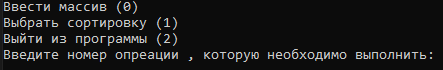


Рисунок 6 – Главное меню

При нажатии на любую другую цифру программа скажет, что не знает такой команды и предложит пользователю выбрать одну из трёх имеющихся.



Рисунок 7 – Пример неверно введённой команды

Рекомендуется сначала ввести массив, затем выбрать сортировку. При нажатии на цифру 0 пользователь сможет ввести необходимое для сортировки количество элементов.



Рисунок 8 – Ввод количества элементов в массиве

Пользователю доступно 4 способа отсортировать массив действительных чисел : сортировкой вставками с линейным поиском (при нажатии на цифру 0), сортировкой вставками с бинарным поиском (при нажатии на цифру 1) , сортировкой Шелла (при нажатии на цифру 2) , поразрядной сортировкой (при нажатии на цифру 3). Также пользователь может вернуться в главное меню, где сможет ввести иное количество элементов в массиве.

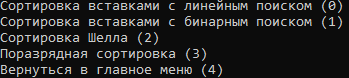


Рисунок 9 – Меню сортировки

Затем программа выведет на экран информацию о том, что массив был успешно отсортирован, а также время работы данной сортировки.



Рисунок 10 – Пример корректной работы программы

Далее пользователь может либо вернуться в главное меню, чтобы сменить массив, либо выбрать другую сортировку, чтобы посмотреть время работы выбранной сортировки и сравнить со временем работы предыдущей.

# Описание программной реализации

void main()- основная функция всей программы.

int menu\_choice(char\* menu[], int ns) – функция принимает на вход указатель на массив значений, которые будет принимать меню, значения. Функция создаёт и выводит меню. Пользователь должен ввести номер операции, которую будет необходимо выполнить. Именно этот номер и возвращает функция.

void Sort\_menu(int id\_2, float\* a, int na, float\* b, float\* c, float\* a\_tmp)-функция создаёт меню сортировок. Принимает на вход отвечающее за выбор операции в меню число, указатель на начало исходного массива, его длину, указатель на начало скопированного массива, указатель на начало временного массива, необходимого для поразрядной сортировки.

void copy\_arr(float\* a, float\* b, int na) – функция принимает на вход указатель на начало исходного массива и указатель на начало массива, который будет являться копией исходного, длину создаваемого массива. Функция копирует элементы одного массива и заполняет ими другой.

int InputArray(float\* a) – функция принимает на вход указатель на начало исходного массива и возвращает количество элементов в массиве. Функция просит пользователя ввести количество элементов в массиве и заполняет этот массив числами с типом данных float с помощью датчика псевдослучайных чисел.

void PrintArray(float\* a, int na) – функция принимает на вход указатель на начало исходного массива и количество элементов в этом массиве. Функция печатает элементы массива на экран.

void correct\_sort(float\* a,int na,float\* с)- функция принимает на вход указатель на начало отсортированного массива, длину массива и указатель на начало исходного массива. Функция сравнивает поэлементно массив а- упорядоченный с помощью реализованных мною сортировок и массив b – упорядоченный с помощью встроенной в стандартную библиотеку языка Си стандартной сортировки qsort() из библиотеки stdlib.

int compare(const void\* a, const void\* b)- функция принимает на вход указатель на элемент a и элемент b , а возвращает результат сравнения числа ac(имеющего такой же указатель как элемент а) и числа bc(имеющего такой же указатель как элемент b).

**Сортировка вставками**

void shift(float\* a, int s, int k) – функция принимает на вход указатель на начало исходного массива, номер первого сдвигаемого элемента s, количество упорядоченных элементов в массиве k. Функция сдвигает элементы a[s],….,a[k- 1] на одну позицию влево.

int place(float\* a, float R, int k) – функция линейного поиска положения элемента в отсортированной части массива. Функция принимает на вход указатель на начало исходного массива, новый элемент R, положение которого необходимо найти, количество упорядоченных элементов в массиве k. Функция возвращает положение элемента R в отсортированной части массива.

int bplace(float\* a, float R, int k)–функция бинарного поиска положения элемента в отсортированной части массива .Функция принимает на вход указатель на начало исходного массива, новый элемент R, положение которого необходимо найти, количество упорядоченных элементов в массиве k. Функция возвращает положение элемента R в отсортированной части массива.

**Сортировка вставками с линейным поиском**

void insert(float\* a, int na) – функция принимает на вход указатель на начало исходного массива и количество элементов в этом массиве. Это и есть функция сортировки вставками с линейным поиском. Для сдвига элементов используется функция void shift(float\* a, int s, int k),а для поиска положения элемента в отсортированной части массива функция линейного поиска int place(float\* a, float R, int k).

**Сортировка вставками с бинарным поиском**

void Binsert(float\* a, int na) – функция принимает на вход указатель на начало исходного массива и количество элементов в этом массиве. Это и есть функция сортировки вставками с бинарным поиском. Для сдвига элементов используется функция void shift(float\* a, int s, int k),а для поиска положения элемента в отсортированной части массива функция бинарного поиска int place(float\* a, float R, int k).

**Сортировка Шелла**

void Shell\_Sort(float\* array, int size)- функция принимает на вход указатель на начало исходного массива и количество элементов в этом массиве. Это и есть функция сортировки Шелла.

**Поразрядная сортировка**

void countByte(unsigned int\* mas, int size, int count[256], int byte) – функция принимает на вход указатель на начало исходного массива ,его длину, массив для подсчёта , рассматриваемый байт. Функция подсчитывает сколько раз какое значение каждого байта числа встретилось, записывая все в массив count[256].

void radix\_sort(unsigned int\* mas, int size, unsigned int\* mas\_tmp) – функция принимает на вход указатель на начало исходного массива ,его длину, указатель на начало временного массива. Эта функция есть поразрядная сортировка для беззнаковых целых чисел.

void radixFloat(float\* mas, int n, float\* mas\_tmp) – функция принимает на вход указатель на начало исходного массива ,его длину, указатель на начало временного массива, необходимого для поразрядной сортировки. Функция при помощи вышеописанных функций сортирует исходный массив. Эта функция есть поразрядная сортировка для типа данных float.

# Подтверждение корректности

Для проверки корректности реализации всех сортировок была использована функция correct\_sort ().

void correct\_sort(float\* a,int na,float\* с)- функция принимает на вход указатель на начало отсортированного массива, длину массива и указатель на начало исходного массива. Функция сравнивает поэлементно массив а- упорядоченный с помощью реализованных мною сортировок и массив с – упорядоченный с помощью встроенной в стандартную библиотеку языка Си стандартной сортировки qsort() из библиотеки stdlib.

int compare(const void\* a, const void\* b)- функция принимает на вход указатель на элемент a и элемент b , а возвращает результат сравнения числа ac(имеющего такой же указатель как элемент а) и числа bc(имеющего такой же указатель как элемент b). Функция устанавливает взаимное отношение двух элементов и является вспомогательной для функции qsort().

Подтверждение корректности происходит следующем образом: сначала исходный массив a[] сортируется одной из реализованных сортировок, затем копия исходного массива(с[]) сортируется с помощью встроенной в библиотеку Си сортировки qsort(). Функция correct\_sort() поэлементно сравнивает два отсортированных массива a и с. В случае несоответствия (a[i] !=с[i]) программа сообщит пользователю об ошибке : "!!!!!!!ОШИБКА!!!!!!! Массив упорядочен НЕКОРРЕКТНО. Проверка правильности была реализована с помощью стандартной сортировки qsort(). !!!!!!!ОШИБКА!!!!!!! "

# Результаты экспериментов

После проведения экспериментов, которые заключались в измерении времени работы каждой сортировки при различных количествах элементов в массиве, можно сказать какую сложность имеет каждый алгоритм. Для этого достаточно разделить время выполнения сортировки массива , выраженное в микросекундах , поскольку если выражать время в секундах мы получим очень малые значения сильно близкие к 0, на среднюю сложность каждого алгоритма и построить график зависимости частного времени и сложности от количества элементов в массиве . По графику сразу будет понятно , что эта зависимость будет стремиться к какой-то константе.

Для экспериментов длина массива изменялась с 1000 элементов до 200000000. На каждом шаге после сортировки массива осуществлялась проверка на корректность. В случае корректного выполнения сортировки выводилось время выполнения алгоритма. По полученным данным была сначала построена таблица, а затем график зависимости времени работы сортировки от количества элементов в массиве. Данный график строился для того, чтобы продемонстрировать какую временную сложность имеет каждая из сортировок.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество элементов в массиве, n | Сортировка вставками с линейным поиском t1, сек | Сортировка вставками с бинарным поиском t2, сек | Сортировка Шелла t3, сек | Поразрядная сортировка t4, сек |
| 1000 | 0,001 | 0,000 | 0,000 | 0,000 |
| 5000 | 0,021 | 0,012 | 0,001 | 0,000 |
| 10000 | 0,076 | 0,042 | 0,002 | 0,001 |
| 15000 | 0,173 | 0,092 | 0,004 | 0,001 |
| 25000 | 0,478 | 0,254 | 0,005 | 0,001 |
| 35000 | 0,938 | 0,488 | 0,01 | 0,001 |
| 50000 | 1,914 | 1,002 | 0,012 | 0,002 |
| 70000 | 3,743 | 1,950 | 0,022 | 0,003 |
| 100000 | 7,666 | 3,976 | 0,027 | 0,005 |
| 150000 | 17,245 | 8,983 | 0,048 | 0,006 |
| 300000 | 71,111 | 36,117 | 0,089 | 0,016 |
| 600000 | 278,864 | 145,583 | 0,201 | 0,024 |
| 1000000 |  | 411,585 | 0,363 | 0,041 |
| 5000000 |  |  | 2,449 | 0,197 |
| 10000000 |  |  | 5,663 | 0,378 |
| 20000000 |  |  | 13,072 | 0,768 |
| 30000000 |  |  | 20,762 | 1,146 |
| 50000000 |  |  | 36,003 | 1,906 |
| 70000000 |  |  | 53,231 | 2,669 |
| 100000000 |  |  | 78,706 | 3,827 |
| 150000000 |  |  |  | 5,718 |
| 200000000 |  |  |  | 7,687 |



По данным экспериментов видно, что сортировка вставками с линейным поиском и сортировка вставками с бинарным поиском показали самый плохой результат. Чем больше элементов, тем больше времени они тратят на сортировку массива, их графики есть правые ветви параболы. Это достаточно очевидно, поскольку у них сложность . По графику можно убедиться, что бинарный поиск улучшает сортировку вставками немного и всё равно имеет большую временную сложность.

Значительно быстрее работает сортировка Шелла. Она начала показывать время отличное от 0 с 5 миллионов элементов, сортировка заняла 2,449 секунд. По графику видно, что её сложность точно не . Линия идёт плавнее ветви параболы, что очень похоже на .Далее можно заметить рост времени у поразрядной сортировки. Она оторвалась от точки 0 на 30 миллионах элементов, затратив всего 1,146 секунд на сортировку всего массива. Очевидно, что её сложность есть O(n).

Теперь в явном виде получим константы, к которым сходятся графики зависимости частного времени работы сортировки в микросекундах и её сложности от количества элементов в массиве. Для этого построим соответствующую таблицу.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество элементов в массиве,n | (t1/o(n^2))\*10^6-время выполнения сортировки вставками линейным поиском делённое на сложность данной сортировки | (t2/o(n^2))\* 10^6-время выполнения сортировки вставками бинарным поиском делённое на сложность данной сортировки | (t3/o(n(log(n))^2))\* 10^6-время выполнения сортировки Шелла делённое на сложность данной сортировки в лучшем случае | (t4/o(n))\*10^6-время выполнения поразрядной сортировки делённое на сложность данной сортировки |
| 1000 | 0,001 | 0 | 0 | 0 |
| 5000 | 0,00084 | 0,00048 | 0,001325 | 0 |
| 10000 | 0,00076 | 0,00042 | 0,001133 | 0,1 |
| 15000 | 0,000769 | 0,000409 | 0,001386 | 0,066667 |
| 25000 | 0,000765 | 0,000406 | 0,000937 | 0,04 |
| 35000 | 0,000766 | 0,000398 | 0,001254 | 0,028571 |
| 50000 | 0,000766 | 0,000401 | 0,000985 | 0,04 |
| 70000 | 0,000764 | 0,000398 | 0,001213 | 0,042857 |
| 100000 | 0,000767 | 0,000398 | 0,000979 | 0,05 |
| 150000 | 0,000766 | 0,000399 | 0,001082 | 0,04 |
| 300000 | 0,00079 | 0,000401 | 0,000896 | 0,053333 |
| 600000 | 0,000775 | 0,000404 | 0,000909 | 0,04 |
| 1000000 |  | 0,000412 | 0,000913 | 0,041 |
| 5000000 |  |  | 0,000989 | 0,0394 |
| 10000000 |  |  | 0,001042 | 0,0378 |
| 20000000 |  |  | 0,001111 | 0,0384 |
| 30000000 |  |  | 0,001121 | 0,0382 |
| 50000000 |  |  | 0,001101 | 0,03812 |
| 70000000 |  |  | 0,001111 | 0,038129 |
| 100000000 |  |  | 0,001114 | 0,03827 |
| 150000000 |  |  |  | 0,03812 |
| 200000000 |  |  |  | 0,038435 |

По данным таблицы можно построить графики зависимости для каждой из сортировок. ****

График сортировки Шелла сходится к числу 0,0011(C~0,0011). Значит, временная сложность этой сортировки действительно равна .



График поразрядной сортировки очевидно сходится к числу 0,04(С~ 0,04). Значит, временная сложность этой сортировки действительно равна O(.).



График сортировки вставками с линейным поиском сходится к 0,0008 (C~0,0008). Значит, временная сложность данной сортировки действительно было O(.



График сортировки вставками с бинарным поиском сходится к 0,0004 (C~0,0004). Значит, временная сложность данной сортировки действительно было O(. Несмотря на бинарный поиск для достаточно большого количества элементов существенного преимущества во времени, по сравнения с сортировкой вставками с линейным поиском, она не даст .

# Заключение

Таким образом, в ходе лабораторной работы, были реализованы на языке программирования Си сортировка вставками с линейным и бинарным поисками, сортировка Шелла и поразрядная сортировка для типа данных float. Программа автоматически проверяет свою корректность, используя встроенную в библиотеку языка Си функцию qsort(), и в случае возникновения ошибки в сортировках выведет на экран сообщение об ошибке. Программа имеет удобное меню. Пользователь без проблем может выбирать сортировки и, при необходимости, менять количество элементов в исходном массиве. Были описаны алгоритмы работы данных сортировок, их программная реализация и проведенные эксперименты для замера и подтверждения их временной сложности. В ходе проведения экспериментов было посчитано время работы каждой сортировки для определённого количества элементов на отрезке от 1000 до 200000000. В соответствии с полученными результатами была заполнена таблица и график зависимости времени работы каждой сортировки от количества сортируемых элементов. Для подтверждения сложности каждой сортировки время работы сортировок было переведено в микросекунды, для удобства, и поделено на их теоретическую сложность. Были построены графики зависимости этого отношения от количества элементов в массиве. Все они стремятся к константе для любой рассматриваемой сортировки. Значит, была верно определена временная сложность каждого алгоритма. Для всех сортировок была получена константа при делении времени выполнения, на сложность алгоритма. Для сортировки вставками с линейным поиском C ≈ 0,0008, для сортировки вставками с бинарным поиском C ≈ 0,0004, для сортировки Шелла C ≈ 0,0011, для поразрядной сортировки C ≈ 0,04. Сортировка вставками как с линейным, так и с бинарным поиском оказалась наименее эффективна на больших массивах, далее идёт сортировка Шелла, которая имеет среднюю эффективность из этих 3-х сортировок, самой лучшей же оказалась поразрядная сортировка.

**Приложение**

**Сортировка вставками:**

//Сдвиг элементов a[s],...,a[k-1] на одну позицию влево

void shift(float\* a, int s, int k) {

// s-номер первого сдвигаемого элемента; k- количество упорядоченных элементов

for (int i = k; i > s; --i)

a[i] = a[i - 1];

}

// Определение местоположения элемента R среди a[0]<=a[1]<=...<=a[k-1](Линейный поиск)

int place(float\* a, float R, int k) {

// R - новый элемент,положение которого необходимо найти

// k - количество упорядоченных элементов в массиве

int s = k, i = 0;

for (; i < k; ++i) {

if (a[i] > R) {

s = i;

break;

}

}

return s;

}

//Бинарный поиск

int bplace(float\* a, float R, int k)

{

int st = 0;

int mid;

int fn = k - 1;

int s;

while (st < fn) {

mid = (st + fn) / 2;

if (R > a[mid]) st = mid + 1;

else fn = mid - 1;

}

if (R > a[st]) s = st + 1;

else s = st;

return s;

}

**Сортировка вставками с линейным поиском:**

//Сортировка вставками с линейным поиском

void insert(float\* a, int na) {

int k, s; float R;

for (k = 1; k < na; ++k) {

R = a[k];

s = place(a, R, k);

shift(a, s, k);

a[s] = R;

}

}

**Сортировка вставками с бинарным поиском:**

//Сортировка вставками с бинарным поиском

void Binsert(float\* a, int na) {

int k, s; float R;

for (k = 1; k < na; ++k) {

R = a[k];

s = bplace(a, R, k);

shift(a, s, k);

a[s] = R;

}

}

**Сортировка Шелла**

// Cортировка Шелла

void Shell\_Sort(float\* array, int size) {

for (int s = size / 2; s > 0; s /= 2) {

for (int i = s; i < size; ++i) {

for (int j = i - s; j >= 0 && array[j] > array[j + s]; j -= s) {

float temp = array[j];

array[j] = array[j + s];

array[j + s] = temp;

}

}

}

}

**Поразрядная сортировка**

//Поразрядная сортировка Radix\_sort

void countByte(unsigned int\* mas, int size, int count[256], int byte) {

unsigned char\* masc;

int bias;

int tmp1, tmp2;

bias = sizeof(unsigned int);

masc = (unsigned char\*)mas;

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

count[i] = 0;

}

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

count[masc[i \* bias + byte]]++;

}

tmp1 = count[0];

count[0] = 0;

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

tmp2 = count[i];

count[i] = count[i - 1] + tmp1;

tmp1 = tmp2;

}

}

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

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

int count[256];

int sizetype = sizeof(unsigned int);

for (int i = 0; i < sizetype; i++) {

countByte(mas, size, count, i);

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

mas\_tmp[count[masc[j \* sizetype + i]]++] = mas[j];

}

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

mas[j] = mas\_tmp[j];

}

}

}

void radixFloat(float\* mas, int n, float\* mas\_tmp) {

int count = 0;

int i = 0;

radix\_sort((unsigned int\*)mas, n, (unsigned int\*)mas\_tmp);

while (i < n) {

if (mas[i] >= 0) {

count++;

}

i++;

}

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

mas\_tmp[i + (n - count)] = mas[i];

for (; i < n; i++)

mas\_tmp[n - (++count)] = mas[i];

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

mas[i] = mas\_tmp[i];

}