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**

**«Прототип файлового менеджера с функцией сортировки по размеру файла»**

**Выполнил**:

студент группы 381903-1

Костюков М.И.

Нижний Новгород

2019

Содержание

[Введение 3](#_Toc26478367)

[1. Постановка задачи 4](#_Toc26478368)

[2. Руководство пользователя 5](#_Toc26478369)

[3.1. Описание структуры программы 8](#_Toc26478370)

[3.2. Описание алгоритмов 9](#_Toc26478371)

[4. Результаты экспериментов 15](#_Toc26478372)

[Заключение 17](#_Toc26478373)

[Литература 18](#_Toc26478374)

[Приложение 19](#_Toc26478375)

# Введение

На сегодняшний день одной из важнейших операций, применяемых каждым пользователем ПК, является сортировка массивов. Она используется для достаточно быстрого и удобного нахождения нужной информации. В условиях повышения сложности решаемых в программировании задач, в которых необходимо работать с большими объёмами информации, куда выгоднее сначала упорядочивать используемый массив данных в нужном порядке, что значительно снизит затраты во времени при поиске какого-либо элемента из этого массива. Упорядочивание данных можно проводить по разным критериям в зависимости от того, с чем собирается работать пользователь или разработчик: упорядочивание по возрастанию/убыванию; в алфавитном порядке; по дате и времени и т.д.

# Постановка задачи

Разработать прототип файлового менеджера с функцией показа файлов в заданном каталоге, упорядоченных по возрастанию/убыванию размера.

Входные данные:

* Путь до директории, в которой необходимо отсортировать содержимое.
* Метод сортировки.

Выходные данные:

* Отсортированный список имен файлов с указанием размера.
* Время сортировки.

Программа должна предоставлять пользователю возможность сменить метод сортировки и повторно формировать выходные данные.

Программа должна реализовывать диалог с пользователем посредством интерфейса, который включает:

* возможность ввода пути до заданного каталога;
* возможность выбора метода сортировки;
* возможность просмотра отсортированного списка файлов с указанием размера.

# Руководство пользователя

В начале программы пользователю предлагается ввести путь до директории, причем нужного формата.

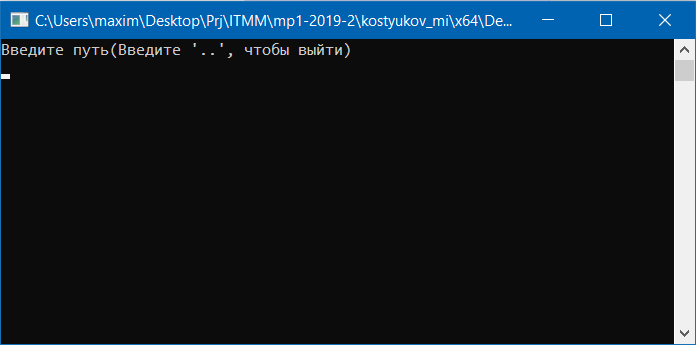


Рис. 1. Начало работы

При неправильном введении пути до директории, предлагает ввести заново.

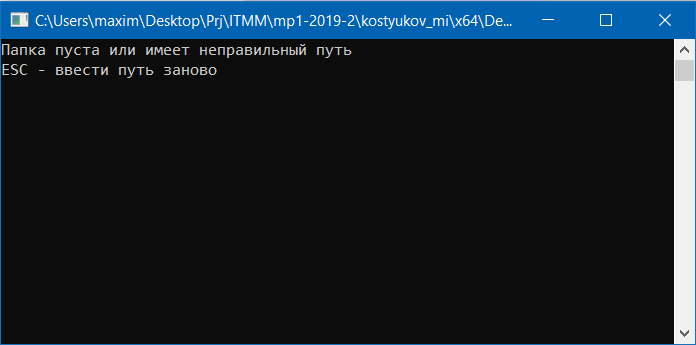


Рис. 2. Неправильный ввод пути

После введения пути, на экран выводится список файлов, находящихся в данной директории, указывается их названия, размеры и количество файлов.

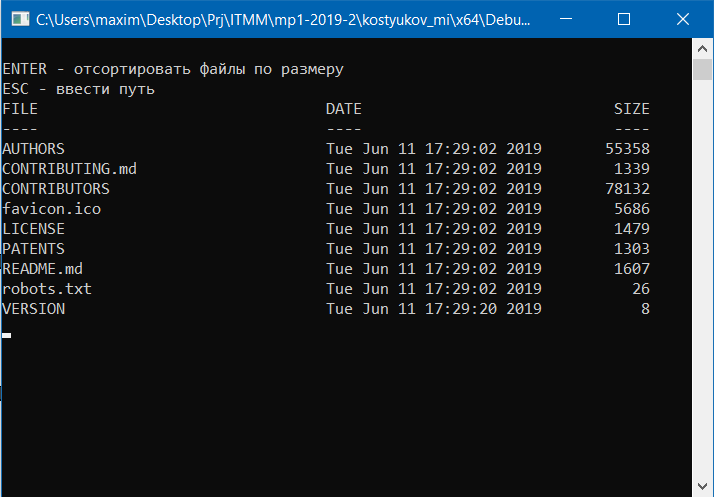


Рис. 3. Вывод списка файлов

При нажатии ENTER на экран выводится список сортировок.

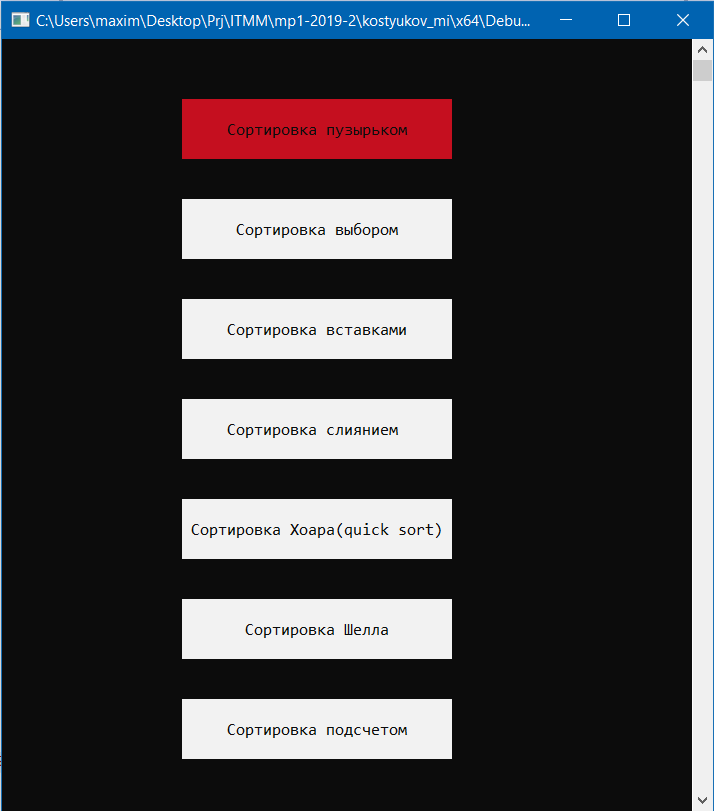


Рис. 4. Выбор сортировки

После ввода выводит отсортированный список файлов с указанием времени работы этого вида сортировки.

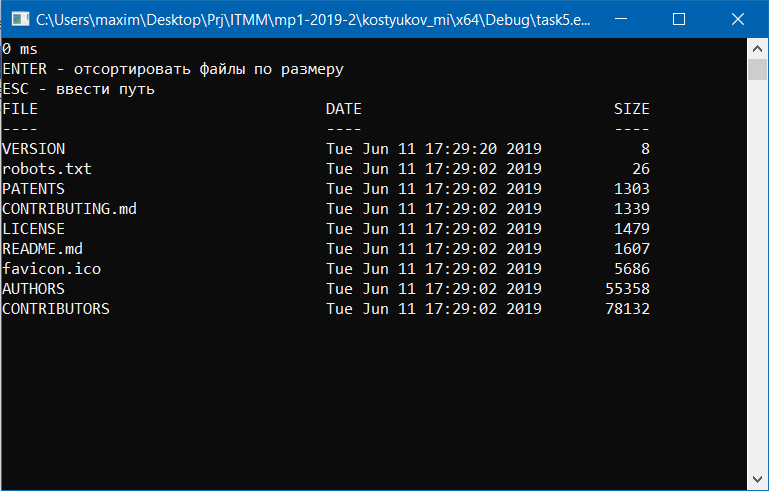


Рис. 5. Вывод отсортированных файлов

Пользователю предлагается продолжить работу или выйти из программы.

Далее пользователь может ввести новый путь, либо отсортировать исходную директорию с помощью другой сортировки. Таким образом, можно продолжать работу в данной программой с разными директориями. При желании пользователь всегда может выйти.

1. **Руководство программиста**

## Описание структуры программы

Программа написана на языке программирования СИ. Состоит из двух файлов: Source.cpp и FileSorts.cpp.

В программе реализованы следующие функции:

1. **Функция «bubleSort»**

Описание: реализация сортировки пузырьком.

Принимает на вход массив типа \_finddata\_t files, размер массива size (long). В результате выполнения функции, массив files становится упорядоченным.

1. **Функция «selectSort»**

Описание: реализация сортировки выбором.

Принимает на вход массив типа \_finddata\_t files, размер массива size (long). В результате выполнения функции, массив files становится упорядоченным.

1. **Функция «insertSort»**

Описание: реализация сортировки вставками.

Принимает на вход массив типа \_finddata\_t files, размер массива size (long). В результате выполнения функции, массив files становится упорядоченным.

1. **Функция «mergeSort»**

Описание: реализация сортировки слиянием.

Принимает на вход массив типа \_finddata\_t files, индекс левой границы begin (long), индекс правой границы end (long). В результате выполнения функции, массив files становится упорядоченным.

1. **Функция «Merge»**

Описание: реализует слияние двух упорядоченных массивов.

Принимает на вход массив типа \_finddata\_t files, индекс левой границы begin (long), индекс середины mid(long), индекс правой границы end (long). В результате получается слитый массив files.

1. **Функция «quickSort»**

Описание: реализация сортировки Хоара.

Принимает на вход массив типа \_finddata\_t files, размер массива size (long). В результате выполнения функции, массив files становится упорядоченным.

1. **Функция «shellSort»**

Описание: реализация сортировки Шелла.

Принимает на вход массив типа \_finddata\_t files, размер массива size (long). В результате выполнения функции, массив files становится упорядоченным.

1. **Функция «Increment»**

Описание: Подсчет приращения для сортировки Шелла по формуле Седжвика.

Принимает на вход массив целого типа inc и размер массива size (long), который нужно отсортировать. В результате возвращает целого типа значение приращения.

1. **Функция «countingSort»**

Описание: реализация сортировки подсчетом.

Принимает на вход массив типа \_finddata\_t files, размер массива size (long). В результате выполнения функции, массив files становится упорядоченным.

## Описание алгоритмов

В программе используются следующие алгоритмы сортировок:

* Пузырьком,
* Выбором,
* Вставками,
* Слиянием,
* Хоара,
* Шелла.

**Сортировка пузырьком:**

Идея метода этой сортировки состоит в проходе по массиву снизу вверх. При этом сравниваются соседние элементы. Если они расположены неправильно, то их меняем местами. После нулевого прохода по массиву (при сортировке по возрастанию) самый «легкий» элемент оказывается «вверху» - отсюда аналогия с пузырьком. Следующий проход делается до второго элемента сверху, и второй по величине элемент будет поставлен на свое место. Далее делаются проходы по все уменьшающейся части массива до тех пор, пока не останется последовательность из одного элемента.

Ниже приведен псевдокод алгоритма (по возрастанию):

**Функция Пузырек** (**тип данных** a (масссив), **цел** size)

{

**цел** i, j;

**тип данных** с;

**цикл для** i от 0 до size-1

**цикл для** j от size-1 до i+1

**если** (a[ j ] < a[ j-1 ]), **то**

{

c = a[ j ];

a[ j ] = a[ j-1 ];

a[ j-1 ] = c;

}

}

**Сортировка выбором:**

Алгоритм этой сортировки заключается в постоянном нахождении минимального или максимального (в зависимости от того как мы хоти отсортировать) элемента в массиве. Найденный элемент меняется позицией с первым. Затем рассмотрению подлежит неотсортированная часть массива, исключая найденный ранее элемент. В этой части также проводится поиск минимального (максимального) элемента и после его нахождения, он меняется позицией со вторым элементом. Такой процесс поиска и обмена местами будет продолжаться, пока массив не будет отсортирован.

Ниже приведен псевдокод алгоритма (по возрастанию):

**Функция Выбором** (**тип данных** a (масссив), **цел** size)

{

**цел** i, j, m;

**тип данных** с;

**цикл для** i от 0 до size-1

{

m = i;

c = a[ i ];

**цикл для** j от i+1 до size-1

**если** (a[ j ] < c), **то**

{

c = a[ j ];

m = j;

}

a[ m ] = a [ i ];

a[ i ] = c;

}

}

**Сортировка вставками:**

Суть алгоритма заключается в том, что на каждом шаге выбирается элемент из неотсортированной части массива, после чего он вставляется на нужную позицию в уже отсортированной части массива. Алгоритм будет работать до тех пор, пока не закончится массив. Выбор элемента из массива можно осуществлять произвольно, но удобнее брать их по порядку.

Ниже приведен псевдокод алгоритма (по возрастанию):

**Функция Вставками** (**тип данных** a (масссив), **цел** size)

{

**цел** i, j;

**тип данных** с;

**цикл для** i от 0 до size-1

{

c = a[ i ];

j = i-1;

**цикл пока** (j >= 0) **и** (c < a[ j ])

{

a[ j+1 ] = a[ j ];

j = j-1;

}

a[ j+1 ] = c;

}

}

**Сортировка слиянием:**

Базируется на принципе «разделяй и властвуй». Массив делится пополам, и каждая его часть снова делится пополам до тех пор, пока он не будет состоять из 1 элемента (считается упорядоченным). Затем происходит слияние каждой упорядоченной части со своей половиной, сохраняя упорядоченность.

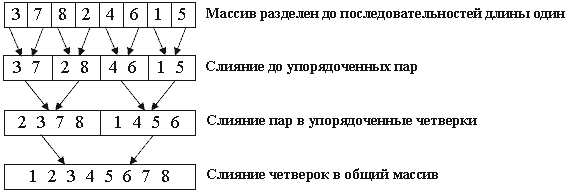


Рис. 6. Пример работы алгоритма на массиве 3 7 8 2 4 6 1 5...

Ниже приведен псевдокод алгоритма:

**Функция Cлиянием** (**тип данных** a (масссив), **цел** l, **цел** r)

{

**цел** mid;

**если** (l < r), **то**

{

mid = (l+r)/2;

Слиянием(a,l,mid);

Слиянием(a,mid+1,r);

Слияние (a,l,mid,r);

}

}

**Функция Cлияние** (**тип данных** a (масссив), **цел** l, **цел** mid, **цел** r)

{

**цел** pos1 = l;

**цел** pos2 = mid+1;

**цел** pos3 = 0;

**тип данных** buf (массив величины r - l+1);

**цикл пока** (pos1 <= mid) **и** (pos2 <= r)

{

**если** (a[ pos1 ] < a[ pos2 ]), **то**

{

buf[ pos3 ] = a[ pos1 ];

pos3 = pos3+1;

pos1 = pos1+1;

}

**иначе**

{

buf [pos3 ] = a[ pos2 ];

pos3 = pos3+1;

pos2 = pos2+1;

}

}

**цикл пока** (pos2 <= r)

{

buf[ pos3 ] = a[ pos2 ];

pos3 = pos3+1;

pos2 = pos2+1;

}

**цикл пока** (pos1 <= mid)

{

buf[ pos3 ] = a[ pos1 ];

pos3 = pos3+1;

pos1 = pos1+1;

}

**цикл для** pos3 от 0 до r-2

a[ pos3 + l ] = buf [ pos3 ];

}

**Сортировка Хоара:**

Метод основан на подходе "разделяй-и-властвуй". Общая схема такова:

1. из массива выбирается некоторый опорный элемент a[i],
2. запускается разделение массива, которое перемещает все ключи, меньшие, либо равные a[i], влево от него, а все ключи, большие, либо равные a[i] - вправо,
3. теперь массив состоит из двух подмножеств, причем левое меньше, либо равно правому
4. для обоих подмассивов, если в подмассиве более двух элементов, рекурсивно запускаем для него ту же процедуру.

В конце получится полностью отсортированная последовательность.

Ниже приведен псевдокод алгоритма:

**Функция Хоара** (**тип данных** a (масссив), **цел** l, **цел** size)

{

**цел** i, j;

**тип данных** buf, mid;

i = l;

j=size-1;

mid = a[ size/2 ];

**цикл до**

{

**цикл пока** (a[ i ] > mid)

i = i+1;

**цикл пока** (a[ j ] < mid)

j = j-1;

**если** (i <= j), **то**

{

buf = a[ i ];

a[ i ] = a[ j ];

a[ j ] = buf;

i = i+1;

j = j-1;

}

}

**пока** (i <= j)

**если** (j > 0), **то**

Хоара(a,l,j+1);

**если** (size > i), **то**

Хоара(a,i,size-i);

}

**Сортировка Шелла:**

Является модификацией сортировки простыми вставками. Идея метода Шелла состоит в сравнении элементов, стоящих не только рядом, но и на определённом расстоянии друг от друга. При данной сортировке сначала сравниваются и сортируются между собой значения, отстоящие один от другого на некотором расстоянии. После этого процедура повторяется для некоторых меньших расстояний, и так продолжается, пока расстояние не будет равно нулю (к этому моменту массив буде уже отсортирован). Единственной характеристикой сортировки Шелла является приращение - расстояние между сортируемыми элементами, в зависимости от прохода. В данной реализации приращение вычисляется по формуле Седжвика.

http://algolist.manual.ru/sort/gif/14_1.gif

Рис. 7. Приращение по формуле Седжвика

Ниже приведен псевдокод алгоритма:

**Функция Шелла** (**тип данных** a (масссив), **цел** size)

{

**цел** i, j, inc, s;

**цел** seq (массив на 40 элементов);

s = приращение (seq, size);

**тип данных** buf;

**цикл пока** (s >= 0)

{

inc = seq[ s ];

s = s-1;

**цикл для** i от inc до size-1

{

buf = a[ i ];

j = j-inc;

**цикл пока** (j >= 0) **и** (buf > a[ j ])

a[ j+inc ] = a[ j ];

j = j-inc;

a[j+inc ] = buf;

}

}

}

**Сортировка подсчетом:**

Это алгоритм сортировки, в котором используется диапазон чисел сортируемого массива (списка) для подсчёта совпадающих элементов. Применение сортировки подсчётом целесообразно лишь тогда, когда сортируемые числа имеют (или их можно отобразить в) диапазон возможных значений, который достаточно мал по сравнению с сортируемым множеством, например, миллион натуральных чисел меньших 1000.

**Функция Шелла** (**тип данных** a (масссив), **цел** size)

{

**цикл для** i от 1 до size

k [i] =0

**цикл для** i от 2 до size

**цикл для** j от 1 до i-1

{

**если** a [i] < a[j] **то**

k [j] = k [j] +1

**иначе**

k [i] = k [i] +1

}

**цикл для** i от 1 до size

b[k[i]+1] = a[i]

}

# Результаты экспериментов

Для определения наиболее быстрой сортировки массивов был проведён эксперимент, в котором измерялось время, за которое каждая сортировка упорядочивает файлы по размеру. Сортировки проводились на разном числе файлов от 314 до 18519. Для каждой сортировки на определенном числе файлов было посчитано среднее значение.

В ходе эксперимента были получены следующие данные по времени сортировки:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Количество элементов в массиве** | **Сортировка Пузырьком** (в сек.) | **Сортировка Выбором** (в сек.) | **Сортировка Вставками** (в сек.) | **Сортировка Слиянием** (в сек.) | **Сортировка Хоара** (в сек.) | **Сортировка Шелла** (в сек.) | **Сортировка подсчетом**  (в сек.) |
| 204 | 0,0026 | 0,0018 | 0,001 | 0,0006 | 0,0004 | 0,0001 | 0,0021 |
| 381 | 0,008 | 0,0063 | 0,0035 | 0,0008 | 0,00057 | 0,00057 | 0,007 |
| 600 | 0,02 | 0,015 | 0,0088 | 0,0014 | 0,0008 | 0,0008 | 0,017 |
| 838 | 0,038 | 0,028 | 0,015 | 0,0022 | 0,0014 | 0,001 | 0,039 |
| 1527 | 0,12 | 0,09 | 0,0518 | 0,0034 | 0,0028 | 0,0026 | — |
| 2669 | 0,4 | 0,29 | 0,196 | 0,0064 | 0,0044 | 0,0046 | — |
| 2974 | 0,507 | 0,36 | 0,2148 | 0,0072 | 0,005 | 0,0058 | — |
| 18519 | 3,1 | 1,84 | 1,24 | 0,0432 | 0,04 | 0,056 | — |

На основе полученных данных были построены графики, отображающие зависимость времени сортировки от количества элементов массива:

Так как сортировки слиянием, Хоара, Шелла работают быстрее, чем остальные три, был построен еще один график, чтобы наглядно убедиться какая из сортировок быстрее.

# Заключение

Анализируя полученные результаты, мы приходим к выводу, что сортировка Хоара является самой быстрой из рассматриваемых сортировок. Но стоит отметить, что данная сортировка является неустойчивой и при определенных входных данных может работать порядка n^2. Самой неэффективной по времени является сортировка подсчетом. Быстрее работают сортировка выбором и вставками. Сортировка слиянием показывает хороший результат, немного уступает Хоара и Шелла, но в отличии от них она устойчивая и дает на любых данных порядка n\*log(n). У слияния в свою очередь есть недостаток: нужно иметь дополнительный массив, что делает его менее выгодным по памяти. Сортировка Шелла работает чуть медленнее Хоара, но в отличие от последней ей не требуется памяти под стек, а также худшее гарантированное время работы сортировки Шелла выше, чем для быстрой сортировки.

Порядок сортировок по их быстроте:

1. Хоара,
2. Шелла,
3. Слиянием,
4. Вставками,
5. Выбором,
6. Пузырек,
7. Подсчетом.

В итоге, для файлового менеджера лучше подойдет сортировка Шелла, ибо более устойчива, чем Хоара и требует меньше памяти, что важно для хорошей работы файлового сортировщика.

# Литература

1. Сайт Algolist. Сортировка выбором – <http://algolist.manual.ru/sort/select_sort.php>.
2. Сайт Algolist. Сортировка пузырьком – <http://algolist.manual.ru/sort/bubble_sort.php>.
3. Сайт Algolist. Сортировка вставками - <http://algolist.manual.ru/sort/insert_sort.php>.
4. Сайт Algolist. Сортировка слиянием - <http://algolist.manual.ru/sort/merge_sort.php>.
5. Сайт Algolist. Сортировка Хоара - <http://algolist.manual.ru/sort/shell_sort.php>.
6. Сайт Algolist. Сортировка Шелла - <http://algolist.manual.ru/sort/shell_sort.php>.

# Приложение

**Текст программы:**

**Файл FileSorts.cpp:**

#include <stdio.h>

#include <stdlib.h>

#include <io.h>

#include <time.h>

#include <string.h>

void bubbleSort(\_finddata\_t\* files, long size)

{

long i, j;

\_finddata\_t temp;

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

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

if (files[j - 1].size > files[j].size)

{

temp = files[j - 1];

files[j - 1] = files[j];

files[j] = temp;

}

}

void selectSort(\_finddata\_t\* files, long size)

{

long i, j, k;

\_finddata\_t temp;

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

{

k = i;

temp = files[i];

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

if (files[j].size < temp.size)

{

k = j;

temp = files[j];

}

files[k] = files[i];

files[i] = temp;

}

}

void quickSort(\_finddata\_t\* files, long size)

{

long i = 0, j = size-1;

unsigned long p;

\_finddata\_t temp;

p = files[(size-1) / 2].size;

while (i <= j)

{

while (files[i].size < p) i++;

while (files[j].size > p) j--;

if (i <= j)

{

temp = files[i];

files[i] = files[j];

files[j] = temp;

i++; j--;

}

};

if (j > 0) quickSort(files, j+1);

if (size-1 > i) quickSort(files+i, size-i);

}

int increment(long inc[], long size)

{

int p1, p2, p3, s;

p1 = p2 = p3 = 1;

s = -1;

do

{

if (++s % 2)

inc[s] = 8 \* p1 - 6 \* p2 + 1;

else

{

inc[s] = 9 \* p1 - 9 \* p3 + 1;

p2 \*= 2;

p3 \*= 2;

}

p1 \*= 2;

} while (3 \* inc[s] < size);

return s > 0 ? --s : 0;

}

void shellSort(\_finddata\_t\* files, long size)

{

long inc, i, j, seq[40];

int s;

s = increment(seq, size);

while (s >= 0)

{

inc = seq[s--];

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

{

\_finddata\_t temp = files[i];

for (j = i - inc; (j >= 0) && (files[j].size > temp.size); j -= inc)

files[j + inc] = files[j];

files[j + inc] = temp;

}

}

}

void insertSort(\_finddata\_t\* files, long n)

{

long i, j;

\_finddata\_t temp;

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

{

temp = files[i];

for (j = i - 1; j >= 0 && files[j].size > temp.size; j--)

files[j + 1] = files[j];

files[j + 1] = temp;

}

}

void countingSort(\_finddata\_t\* files, long n)

{

long max = 0;

long i = 0;

\_finddata\_t\* sortedfiles = (\_finddata\_t\*)malloc(n \* sizeof(\_finddata\_t));

memcpy(sortedfiles, files, n \* sizeof(\_finddata\_t));

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

{

if (files[i].size > max)

max = files[i].size;

}

max++;

int\* C = (int\*)malloc(max \* sizeof(int));

memset(C, 0, max \* sizeof(int));

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

{

C[files[i].size]++;

}

int c = 0;

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

{

int t = C[i];

C[i] = c;

c += t;

}

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

{

sortedfiles[C[files[i].size]] = files[i];

C[files[i].size]++;

}

memmove(files, sortedfiles, n \* sizeof(\_finddata\_t));

free(C);

free(sortedfiles);

}

void merge(\_finddata\_t\* files, long begin, long split, long end)

{

long pos1 = begin, pos2 = split + 1, pos3 = 0;

\_finddata\_t\* temp = (\_finddata\_t\*)malloc((end - begin + 1) \* sizeof(\_finddata\_t));

while ((pos1 <= split) && (pos2 <= end))

{

if (files[pos1].size < files[pos2].size)

temp[pos3++] = files[pos1++];

else

temp[pos3++] = files[pos2++];

}

while (pos2 <= end)

temp[pos3++] = files[pos2++];

while (pos1 <= split)

temp[pos3++] = files[pos1++];

for (pos3 = 0; pos3 < end - begin + 1; pos3++)

{

files[begin + pos3] = temp[pos3];

}

free(temp);

}

void mergeSort(\_finddata\_t\* files, long begin, long end)

{

long split;

if (begin < end)

{

split = (begin + end) / 2;

mergeSort(files, begin, split);

mergeSort(files, split + 1, end);

merge(files, begin, split, end);

}

}

void \_countingSort(\_finddata\_t\* files, long size)

{

clock\_t t = clock();

countingSort(files, size);

printf("%d ms", clock() - t);

}

void \_mergeSort(\_finddata\_t\* files, long size)

{

clock\_t t = clock();

mergeSort(files, 0, size - 1);

printf("%d ms", clock() - t);

}

void \_shellSort(\_finddata\_t\* files, long size)

{

clock\_t t = clock();

shellSort(files, size);

printf("%d ms", clock() - t);

}

void \_quickSort(\_finddata\_t\* files, long size)

{

clock\_t t = clock();

quickSort(files, size);

printf("%d ms", clock() - t);

}

void \_insertSort(\_finddata\_t\* files, long size)

{

clock\_t t = clock();

insertSort(files, size);

printf("%d ms", clock() - t);

}

void \_selectSort(\_finddata\_t\* files, long size)

{

clock\_t t = clock();

selectSort(files, size);

printf("%d ms", clock() - t);

}

void \_bubbleSort(\_finddata\_t\* files, long size)

{

clock\_t t = clock();

bubbleSort(files, size);

printf("%d ms", clock() - t);

}

void(\*Sorts[7])(\_finddata\_t\* files, long len) = {

\_bubbleSort,

\_selectSort,

\_insertSort,

\_mergeSort,

\_quickSort,

\_shellSort,

\_countingSort,

};

**Файл Source.cpp:**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <io.h>

#include <time.h>

#include <conio.h>

#include "Console.cpp"

#include "FileSorts.cpp"

#include <string.h>

\_finddata\_t files[10000];

int size;

long readfolder(char\* path, \_finddata\_t\* files)

{

\_finddata\_t file;

intptr\_t hFile;

long count = 0;

if ((hFile = \_findfirst(path, &file)) != -1L)

{

do

{

if (!(file.attrib & \_A\_SUBDIR))

{

files[count] = file;

count++;

}

if (count > 1000)

break;

} while (\_findnext(hFile, &file) == 0);

return count;

}

else

return 0;

}

void printFiles(\_finddata\_t\* files, long size)

{

printf("\nENTER - отсортировать файлы по размеру\nESC - ввести путь\n");

printf("FILE %30c DATE %24c SIZE\n", ' ', ' ');

printf("---- %30c ---- %24c ----\n", ' ', ' ');

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

{

char buffer[30];

ctime\_s(buffer, \_countof(buffer), &files[i].time\_write);

if (files[i].size > 0)

printf("%-35.35s %.24s %10lu \n", files[i].name, buffer, files[i].size);

else

printf("%-35.35s %.24s\n", files[i].name, buffer);

}

}

int sortFiles(int menustate)

{

Sorts[menustate](files, size);

printFiles(files, size);

return 1;

}

void chooseSort()

{

MenuElement menuSort[] = {

{"Сортировка пузырьком" , sortFiles},

{"Сортировка выбором" , sortFiles},

{"Сортировка вставками" , sortFiles},

{"Сортировка слиянием " , sortFiles},

{"Сортировка Хоара(quick sort)" , sortFiles},

{"Сортировка Шелла" , sortFiles},

{"Сортировка подсчетом" , sortFiles},

};

Switcher(menuSort, 7);

}

void main(void)

{

setlocale(LC\_ALL, "Rus");

HideCursor();

gotoxy(0, 0);

char path[10000] ="c:\\temp\\\*";

char input[10000];

while (true)

{

printf("Введите путь(Введите '..', чтобы выйти)\n");

scanf("%s", path);

if (!(strcmp(path, "..")))

break;

strcat(path, "\*");

clrscr();

size = readfolder(path, files);

if (size > 0)

{

printFiles(files, size);

while (true)

{

char kb = 0;

if (\_kbhit())

{

kb = \_getch();

if (kb == 13)

{

clrscr();

size = readfolder(path, files);

chooseSort();

}

if (kb == 27)

{

clrscr();

break;

}

}

}

}

else

{

printf("Папка пуста или имеет неправильный путь\nESC - ввести путь заново");

char kb = 0;

while (true)

if (\_kbhit())

{

kb = \_getch();

if (kb == 27)

break;

}

}

}

}