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**

**«Моделирование работы сортировщика файлов ОС»**

**Выполнил**:

студент группы 381803-1

Мишин И.А.

**Проверил**:

доцент кафедры МОСТ, к.т.н.,

Сысоев А.В.

Нижний Новгород

2018

Содержание

[Введение 3](#_Toc531089541)

[1. Постановка задачи 4](#_Toc531089542)

[2. Руководство пользователя 5](#_Toc531089543)

[3. Руководство программиста 7](#_Toc531089544)

[3.1. Описание структуры программы 7](#_Toc531089545)

[3.2. Описание алгоритмов 9](#_Toc531089546)

[4. Результаты экспериментов 15](#_Toc531089547)

[Заключение 17](#_Toc531089548)

[Литература 18](#_Toc531089549)

[Приложение 19](#_Toc531089550)

# 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

Целью данной лабораторной работы является создание прототипа файлового менеджера, исследование 7 методов сортировки (сортировка пузырьком, сортировка выбором, сортировка вставками, сортировка подсчётом, сортировка Шелла, быстрая сортировка, сортировка слиянием) на их быстродействие и эффективность при работе по сортировке в папках с различным количеством файлов.

# Руководство пользователя

В начале работы программы пользователю выводится приветствующая фраза (рис. 1). Для перехода к вводу пути до папки, в которой предполагается сортировка, пользователю предлагается нажать любую клавишу.

Рис.1 Приветствие

На следующем экране пользователю предлагается ввести путь до папки, в которой предполагается сортировка (рис. 2).

Рис. 2 Ввод пути до папки

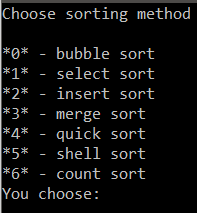
После ввода пути, если указанная папка существует, то пользователь выбирает один из шести предложенных методов сортировки (рис.3).

Рис.3 Методы сортировки

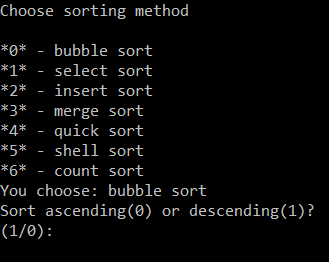
После выбора метода сортировки предлагается выбрать режим сортировки: по возрастанию или по убыванию (рис.4).

Рис.4 Выбор режима сортировки

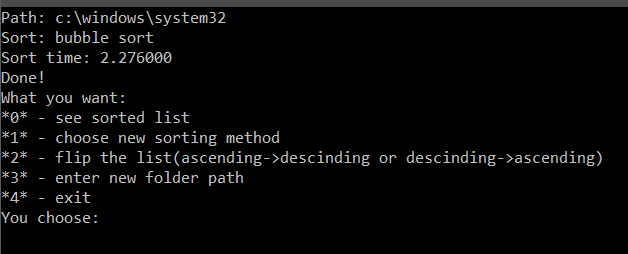
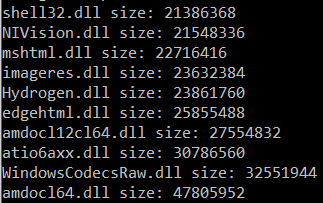
Затем на следующем экране, пользователю предоставляется информация о пути к папке, который он указал, методе сортировки, времени сортировки. Также ему предлагается выбрать один из четырех пунктов меню: увидеть отсортированную папку, выбрать новый метод сортировки, выбрать противоположный режим сортировки, указать новый путь к новой папке, выйти из программы (рис.5).

Рис.5 Основное меню

Соответственно выборы могут появиться такие экраны:

1. Увидеть отсортированную папку.

**Пример:**

Выводятся имена файлов и их размеры в соответствии в выбранным режимом сортировки. Чтобы вернуться к меню, пользователь должен нажать любую клавишу.

1. Выбрать другой метод сортировки.

Пользователь возвращается к экрану с рис.4 и далее продолжает идти по цепочке действий.

1. Перевернуть список.

Меняет режим сортировки с возрастающего на убывающий и наоборот, затем возвращается к меню

1. Ввести новый путь к папке.

Возвращает пользователя к экрану с рис.2 и далее продолжает идти по цепочки действий.

1. Выход из программы.

# Руководство программиста

## Описание структуры программы

Программа состоит из трех файлов: главный файл (Source.cpp), файл сортировками (Sorts.cpp) и заголовочный файл (Header.h).

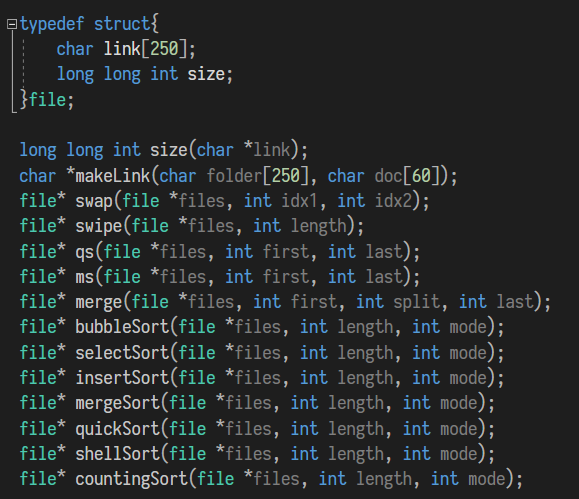
В файле заголовков находятся названия всех функций, используемых программой, также там описывается основная структура file, которая включает в себя два поля: имя файла и его размер (рис.6).

Рис.6 Заголовочный файл

В главном файле происходят основные действия программы: проверка на существование папки, которую указал пользователь, обработка файлов в папке, получение их размеров в байтах, выбор сортировки и вывод элементов меню.

В главном файле важные функции:

**-long long int size(char \*link)**

На вход подается путь к файлу. Функция возвращает размер файла в байтах.

**-char \*makeLink(char folder[250], char doc[60])**

На вход подается путь к папке и имя файла. Функция возвращает путь до конкретного файла.

**-void chooseSort(int sort, int length, int mode)**

На вход подается номер выбранной сортировки, количество файлов и режим сортировки. Процедура сортирует массив структур, также выводит время сортировки.

**-void printSort(char sort)**

Вспомогательная процедура для конвертирования номера выбранной сортировки в ее название. На вход подается номер сортировки. Процедура выводит ее название.

В файле Sorts.cpp представлены семь сортировок, каждая вызывается отдельной функцией на вход подается сортируемый массив, его длина и режим сортировки, функция возвращает отсортированный массив.

Также в этом файле находятся две вспомогательные функции:

**-file\* swap(file \*files, int idx1, int idx2)**

Функция, которая меняет два элемента массива местами.

На вход подается массив и два индекса элементов, которые требуется поменять местами. Функция возвращает массив, с обменянными элементами.

**-file\* swipe(file \*files, int length)**

Функция, которая разворачивает массив, то есть из массива, упорядоченного по возрастанию, делает массив, упорядоченный по убыванию и наоборот

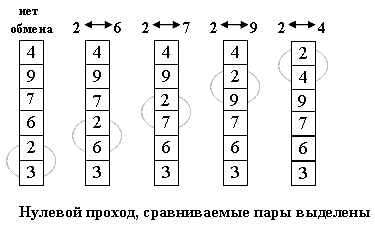
В остальных функциях реализованы сортировки, об их реализации будет сказано в следующем пункте, а код будет представлен в приложении.

## Описание алгоритмов

*Сортировка пузырьком*

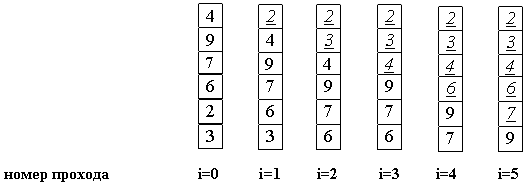
**-file\* bubbleSort(file \*files, int length, int mode)**

Идея сортировки: шаг сортировки состоит в проходе снизу вверх по массиву. По пути просматриваются пары соседних элементов. Если элементы некоторой пары находятся в неправильном порядке, то меняем их местами.



После нулевого прохода по массиву "вверху" оказывается самый "легкий" элемент - отсюда аналогия с пузырьком. Следующий проход делается до второго сверху элемента, таким образом второй по величине элемент поднимается на правильную позицию...

Делаем проходы по все уменьшающейся нижней части массива до тех пор, пока в ней не останется только один элемент. На этом сортировка заканчивается, так как последовательность упорядочена по возрастанию.



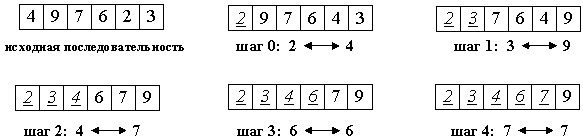
Среднее число сравнений и обменов имеют квадратичный порядок роста: Theta(n2), отсюда можно заключить, что алгоритм пузырька очень медленен и малоэффективен.

*Сортировка выбором*

**-file\* selectSort(file \*files, int length, int mode)**

Идея метода состоит в том, чтобы создавать отсортированную последовательность путем присоединения к ней одного элемента за другим в правильном порядке.

Будем строить готовую последовательность, начиная с левого конца массива. Алгоритм состоит из n последовательных шагов, начиная от нулевого и заканчивая (n-1)-м.

****На i-м шаге выбираем наименьший из элементов a[i] ... a[n] и меняем его местами с a[i]. Последовательность шагов при n=5 изображена на рисунке ниже.

Вне зависимости от номера текущего шага i, последовательность a[0]...a[i] (выделена курсивом) является упорядоченной. Таким образом, на (n-1)-м шаге вся последовательность, кроме a[n] оказывается отсортированной, а a[n] стоит на последнем месте по праву: все меньшие элементы уже ушли влево.

Для нахождения наименьшего элемента из n+1 рассматримаемых алгоритм совершает n сравнений. С учетом того, что количество рассматриваемых на очередном шаге элементов уменьшается на единицу, общее количество операций:

n + (n-1) + (n-2) + (n-3) + ... + 1 = 1/2 \* ( n2+n ) = Theta(n2).

Таким образом, так как число обменов всегда будет меньше числа сравнений, время сортировки растет квадратично относительно количества элементов. Алгоритм не использует дополнительной памяти: все операции происходят "на месте".

*Сортировка вставками*

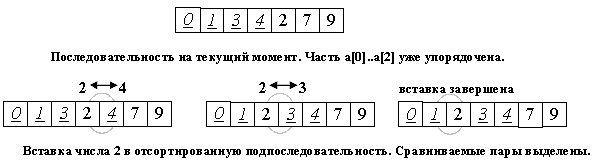
**-file\* insertSort(file \*files, int length, int mode)**

Сортировка простыми вставками в чем-то похожа на вышеизложенные методы. Аналогичным образом делаются проходы по части массива, и аналогичным же образом в его начале "вырастает" отсортированная последовательность...

Однако в сортировке пузырьком или выбором можно было четко заявить, что на i-м шаге элементы a[0]...a[i] стоят на правильных местах и никуда более не переместятся. Здесь же подобное утверждение будет более слабым: последовательность a[0]...a[i] упорядочена. При этом по ходу алгоритма в нее будут вставляться (см. название метода) все новые элементы.

Будем разбирать алгоритм, рассматривая его действия на i-м шаге. Как говорилось выше, последовательность к этому моменту разделена на две части: готовую a[0]...a[i] и неупорядоченную a[i+1]...a[n].

На следующем, (i+1)-м каждом шаге алгоритма берем a[i+1] и вставляем на нужное место в готовую часть массива. Поиск подходящего места для очередного элемента входной последовательности осуществляется путем последовательных сравнений с элементом, стоящим перед ним.

В зависимости от результата сравнения элемент либо остается на текущем месте(вставка завершена), либо они меняются местами и процесс повторяется.

Таким образом, в процессе вставки мы "просеиваем" элемент x к началу массива, останавливаясь в случае, когда

1. найден элемент, меньший x или
2. достигнуто начало последовательности.

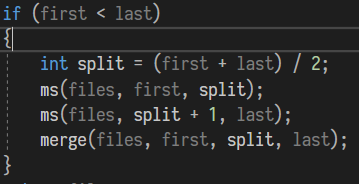
Аналогично сортировке выбором, среднее, а также худшее число сравнений и пересылок оцениваются как Theta(n2), дополнительная память при этом не используется.

*Сортировка слиянием*

**-file\* mergeSort(file \*files, int length, int mode)**

**-file\* ms(file \*files, int first, int last)**

**-file\* merge(file \*files, int first, int split, int last)**

Идея сортировки состоит в том, что изначальный массив делится пополам, каждая половина рекурсивно передается в эту же функцию, пока в подмассиве не остается один элемент.

Функция merge на месте двух упорядоченных массивов a[first]...a[split] и a[split+1]...a[last] создает единый упорядоченный массив a[first]...a[last].

*merge (упорядоченные последовательности A, B, буфер C) {*

*пока A и B не пусты {*

*сравнить первые элементы A и B*

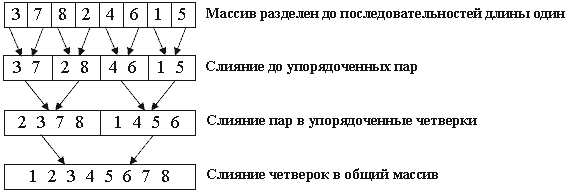
*переместить наименьший в буфер*

*}*

*если в одной из последовательностей еще есть элементы*

*дописать их в конец буфера, сохраняя имеющийся порядок*

*}*

Пример работы алгоритма на массиве {3 7 8 2 4 6 1 5} :

.

*Быстрая сортировка(Хоара)*

**-file\* quickSort(file \*files, int length, int mode)**

**-file\* qs(file \*files, int first, int last)**

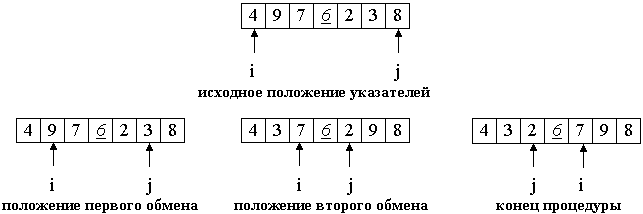
Метод основан на подходе "разделяй-и-властвуй". Общая схема такова:

1. из массива выбирается некоторый опорный элемент a[i],
2. запускается процедура разделения массива, которая 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 правого,  
   http://algolist.manual.ru/sort/gif/22.gif
4. для обоих подмассивов: если в подмассиве более двух элементов, рекурсивно запускаем для него ту же процедуру.

В конце получится полностью отсортированная последовательность.

На входе массив a[0]...a[N] и опорный элемент p, по которому будет производиться разделение.

1. Введем два указателя: i и j. В начале алгоритма они указывают, соответственно, на левый и правый конец последовательности.
2. Будем двигать указатель i с шагом в 1 элемент по направлению к концу массива, пока не будет найден элемент a[i] >= p. Затем аналогичным образом начнем двигать указатель j от конца массива к началу, пока не будет найден a[j] <= p.
3. Далее, если i <= j, меняем a[i] и a[j] местами и продолжаем двигать i,j по тем же правилам...
4. Повторяем шаг 3, пока i <= j.

Рассмотрим работу процедуры для массива a[0]...a[6] и опорного элемента p = a[3].

*Сортировка Шелла*

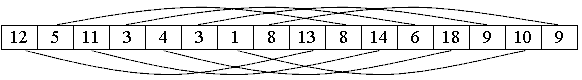
**-** **file\* shellSort(file \*files, int length, int mode)**

Сортировка Шелла является довольно интересной модификацией алгоритма сортировки простыми вставками.

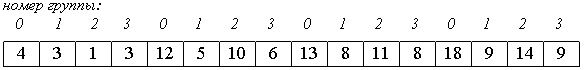
Рассмотрим следующий алгоритм сортировки массива a[0].. a[15].

http://algolist.manual.ru/sort/gif/11.gif

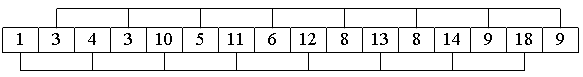
1. Вначале сортируем простыми вставками каждые 8 групп из 2-х элементов (a[0], a[8[), (a[1], a[9]), ... , (a[7], a[15]).



2. Потом сортируем каждую из четырех групп по 4 элемента (a[0], a[4], a[8], a[12]), ..., (a[3], a[7], a[11], a[15]).

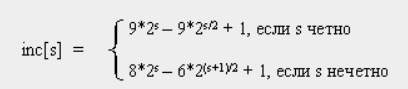


В нулевой группе будут элементы 4, 12, 13, 18, в первой - 3, 5, 8, 9 и т.п.

3. Далее сортируем 2 группы по 8 элементов, начиная с (a[0], a[2], a[4], a[6], a[8], a[10], a[12], a[14]).

**http://algolist.manual.ru/sort/gif/15.gif**4. В конце сортируем вставками все 16 элементов.

Единственной характеристикой сортировки Шелла является приращение - расстояние между сортируемыми элементами, в зависимости от прохода. В конце приращение всегда равно единице - метод завершается обычной сортировкой вставками, но именно последовательность приращений определяет рост эффективности. В программе приращение рассчитывается в отдельной функции

**- int increment(long inc[], long size)** , по формуле Селджвика:

При использовании таких приращений среднее количество операций: O(n7/6), в худшем случае - порядка O(n4/3).

*Сортировка Подсчетом*

**-** **file\* countingSort(file \*files, int length, int mode)**

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, поэтому для решения задачи сортировки файлов в папке он малопригоден и в программе стоит ограничение на его использование, с помощью него можно отсортировать папку с текстовыми файлами небольшого размера.

Помимо входного массива A потребуется два вспомогательных массива — C[0..k - 1] для счётчика и B[0..n - 1] для отсортированного массива. Сначала следует заполнить массив C нулями, и для каждого A[i] увеличить C[A[i]] на 1. Далее подсчитывается количество элементов меньших или равных k-1. Для этого каждый C[j], начиная с C[1], увеличивают на C[j - 1]. Таким образом в последней ячейке будет находиться количество элементов от 0 до k-1 существующих во входном массиве. На последнем шаге алгоритма читается входной массив с конца, значение C[A[i]] уменьшается на 1 и в каждый B[C[A[i]]] записывается A[i]. Алгоритм устойчив.

StableCountingSort

for i = 0 to k - 1

C[i] = 0;

for i = 0 to n - 1

C[A[i]] = C[A[i]] + 1;

for j = 1 to k - 1

C[j] = C[j] + C[j - 1];

for i = n - 1 to 0

C[A[i]] = C[A[i]] - 1;

B[C[A[i]]] = A[i];

# Результаты экспериментов

В небольших папках все сортировки работают очень быстро.

Эксперименты проводились на папке c:\windows\system32

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Пузырьком | Выбором | Вставками | Слиянием | Быстрая | Шелла | Подсчетом |
| 1.774000 | 0.055000 | 0.621000 | 0.041000 | 0.006000 | 0.013000 | --------------- |
| 1.964000 | 0.069000 | 0.422000 | 0.047000 | 0.007000 | 0.008000 |  |
| 1.830000 | 0.045000 | 0.377000 | 0.031000 | 0.006000 | 0.008000 |  |
| 1.946000 | 0.058000 | 0.355000 | 0.030000 | 0.005000 | 0.009000 |  |
| 1.697000 | 0.062000 | 0.366000 | 0.030000 | 0.006000 | 0.010000 |  |
| 1.584000 | 0.056000 | 0.432000 | 0.037000 | 0.007000 | 0.010000 |  |
| 1.822000 | 0.055000 | 0.373000 | 0.032000 | 0.005000 | 0.012000 |  |

# Заключение

Поставленная задача была выполнена. Реализован прототип файлового менеджера с функцией показа файлов в заданном каталоге, упорядоченных по возрастанию/убыванию размера. Программа выполняет сортировку следующими алгоритмами: Сортировка пузырьком, сортировка выбором, сортировка вставками, сортировка слиянием, сортировка Шелла быстрая сортировка (Хоара), сортировка подсчетом. Был проведен эксперимент, в котором замерялось время работы различных алгоритмов, сравнение скорости их работы. Результаты показали, что из используемых сортировок быстрее всех работает сортировка Хоара, медленнее всех сортировка пузырьком.

# Литература

1. Кнут Д. Э. Искусство программирования. Том 3. Сортировка и поиск = The Art of Computer Programming. Volume 3. Sorting and Searching / под ред. В. Т. Тертышного (гл. 5) и И. В. Красикова (гл. 6). – 2-е изд. – Москва: Вильямс, 2007. – Т. 3. – 832 с.
2. Сайт Algolist. Сортировка выбором – <http://algolist.manual.ru/sort/select_sort.php>.
3. Сайт Algolist. Сортировка вставками – <http://algolist.manual.ru/sort/bubble_sort.php>.
4. Сайт Algolist. Сортировка слиянием – <http://algolist.manual.ru/sort/bubble_sort.php>.
5. Сайт Algolist. Сортировка Хоара – <http://algolist.manual.ru/sort/bubble_sort.php>.
6. Сайт Algolist. Сортировка Шелла – <http://algolist.manual.ru/sort/bubble_sort.php>.
7. Сайт Википедия. Сортировка Подсчетом - <https://ru.wikipedia.org/wiki/Сортировка_подсчётом>

# Приложение

Нахождение папки:

HANDLE h = FindFirstFile(strs, &FindFileData);

int k = 0, flag = 0;

if (h != INVALID\_HANDLE\_VALUE)

{

while (FindNextFile(h, &FindFileData))

k++;

Нахождение размера:

long long int size(char \*link)

{

struct \_stat st;

int res = \_stat(link, &st);

if (res == 0)

return st.st\_size;

}

Сортировки:

file\* bubbleSort(file \*files, int length, int mode)

{

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

{

for (int j = length - 1; j > i; j--)

{

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

{

files = swap(files, j, j - 1);

}

}

}

if (mode == 1)

files = swipe(files, length);

return files;

}

file\* selectSort(file \*files, int length, int mode)

{

long long int max = -2;

int k;

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

{

max = -2;

for (int j = i; j < length; j++)

{

if (files[j].size > max)

{

max = files[j].size;

k = j;

}

}

files = swap(files, i, k);

}

if (mode == 0)

files = swipe(files, length);

return files;

}

file\* insertSort(file \*files, int length, int mode)

{

int j;

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

{

file x = files[i];

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

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

files[j + 1] = x;

}

if (mode == 1)

files = swipe(files, length);

return files;

}

file\* mergeSort(file \*files, int length, int mode)

{

ms(files, 0, length - 1);

if (mode == 1)

files = swipe(files, length);

return files;

}

file\* ms(file \*files, int first, int last)

{

if (first < last)

{

int split = (first + last) / 2;

ms(files, first, split);

ms(files, split + 1, last);

merge(files, first, split, last);

}

return files;

}

file\* merge(file \*files, int first, int split, int last)

{

file \*sorted;

sorted = (file \*)calloc(last - first + 1, sizeof(file));

int i = first, j = split + 1;

for (int k = 0; k <= last - first; k++)

if ((j > last) || ((i <= split) && (files[i].size < files[j].size)))

sorted[k] = files[i++];

else

sorted[k] = files[j++];

for (int k = first; k <= last; k++)

files[k] = sorted[k - first];

return files;

}

file\* quickSort(file \*files, int length, int mode)

{

files = qs(files, 0, length-1);

if (mode == 1)

files = swipe(files, length);

return files;

}

file\* qs(file \*files, int first, int last)

{

int i = first;

int j = last;

file x = files[(first + last) / 2];

do

{

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

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

if (i <= j)

{

files = swap(files, i, j);

i++; j--;

}

} while (i <= j);

if (last > i) qs(files, i, last);

if (j > first) qs(files, first , j);

return files;

}

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;

}

file\* shellSort(file \*files, int length, int mode)

{

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

int s;

s = increment(seq, length);

while (s >= 0)

{

inc = seq[s--];

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

{

file 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;

}

}

if (mode == 1)

files = swipe(files, length);

return files;

}

file\* countingSort(file \*files, int length, int mode)

{

file \*sorted;

sorted = (file \*)calloc(length, sizeof(file));

int max = 0, \*counts;

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

{

if (files[i].size > max)

max = files[i].size;

}

if (max > 1000 || length < max)

{

printf(" (This type of sort isn't suitable for this folder) ");

return 0;

}

counts = (int \*)calloc(max + 2, sizeof(int));

for (int i = 0; i < max + 2; i++)

counts[i] = 0;

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

++counts[files[i].size + 1];

if (counts[0] != 0)

counts[0] = counts[0] \* (-1);

for (int i = 1; i < max + 2; i++)

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

for (int i = length-1; i >= 0; i--)

{

counts[files[i].size + 1] -= 1;

sorted[counts[files[i].size + 1]] = files[i];

}

if (mode == 1)

sorted = swipe(sorted, length);

return sorted;

}