Министерство науки и высшего образования РФ

Федеральное государственное автономное образовательное учреждение высшего образования "Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского" (ННГУ)

Институт информационных технологий, математики и механики

Отчет по учебной практике

**Прототип файлового менеджера с функцией показа файлов в заданном каталоге**

Выполнил:

студент гр. 381806-1

Сидорова А.К.

Проверил:

Доцент кафедры МОСТ ИТММ

Кустикова В.Д.

Нижний Новгород

2018 г.

**Содержание**

[Введение 2](#_Toc533343447)

[1. Постановка задачи 3](#_Toc533343448)

[2. Руководство пользователя 4](#_Toc533343449)

[3. Руководство программиста 7](#_Toc533343450)

[3.1. Структура программы 7](#_Toc533343451)

[3.2. Описание алгоритмов 7](#_Toc533343452)

[3.3. Описание функций 15](#_Toc533343453)

[Заключение 20](#_Toc533343454)

# Введение

Каждый день пользователям персонального компьютера приходится работать с большим количеством файлов, приложений или документов, которые хранятся в различных папках. Пытаясь найти тот или иной нужный документ, пользователь затрачивает огромное количество времени, когда во всех папках файлы разбросаны хаотично и беспорядочно. Для того чтобы уменьшить данное время, существуют различные виды сортировок. Например, документы можно отсортировать по названиям, по их расширениям или по их размерам. Отсортировав таким образом файлы, пользователь сможет в разы быстрее найти нужный ему документ.

В данной практической работе будет рассмотрен прототип файлового менеджера, который сможет, опираясь на размеры документов, отсортировать по возрастанию или по убыванию, а также с помощью различных видов сортировок файлы, которые хранятся в той папке, к которой укажет путь пользователь. Кроме того, пользователь сможет после сортировки увидеть количество времени, затраченное на работу алгоритма.

# Постановка задачи

В данной практической работе необходимо было реализовать прототип файлового менеджера, который сортирует файлы относительно их размеров по возрастанию или по убыванию в заданном каталоге с помощью нескольких сортировок: «пузырьковой» сортировки, сортировки выбором, сортировки вставками, сортировки подсчетом, быстрой сортировки и сортировки слиянием. А также данный прототип должен выводить список файлов и их размеров. Кроме того, после сортировки пользователь сможет увидеть количество времени, затраченного на данную сортировку, а также сменить ее вид.

На вход поступает путь до того каталога, в котором необходима сортировка файлов, номер самой сортировки, который отвечает за вид сортировки («пузырьковая», выбором, вставками, подсчетом, быстрая и слиянием) и способ сортировки, то есть либо возрастание, либо убывание.

На выходе пользователь получает сначала получает неотсортированный список файлов с их размерами, а после введенных данных уже отсортированный список по выбранным раннее параметрам, а также время, затраченное на данную сортировку файлов.

# Руководство пользователя

Рассмотрим один из вариантов использования программы, которая была разработана в ходе выполнения данной практической работы.

После запуска программы перед пользователем появится окно (Рис. 1), в котором содержится название программы *«Quick Sort»* и сообщение, в котором просят ввести путь до того каталога, в котором необходимо провести сортировку файлов по их размерам.

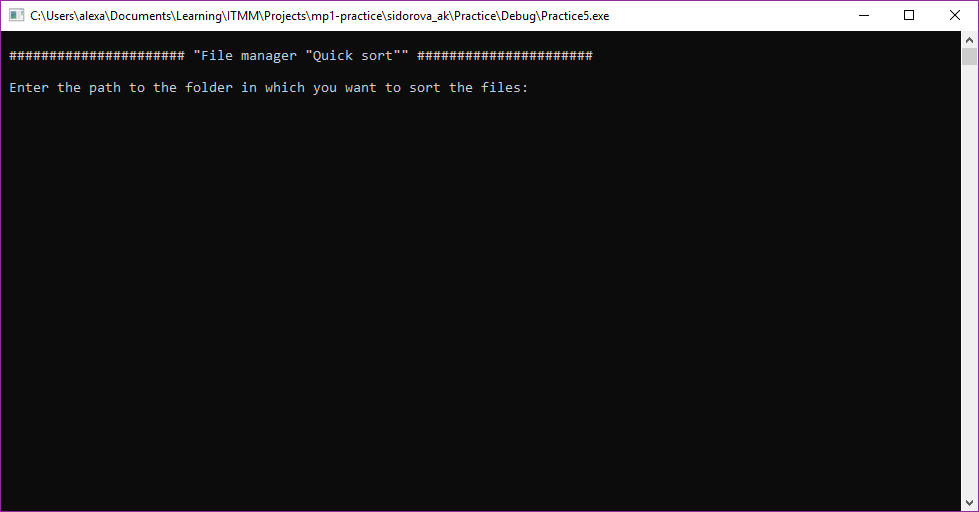


Рис. 1

В предложенном поле необходимо ввести путь до нужного каталога. Например, требуется отсортировать файлы в каталоге «*C:\Windows\System32»*. В таком случае введем данный путь в поле и нажмем на Enter (Рис. 2).

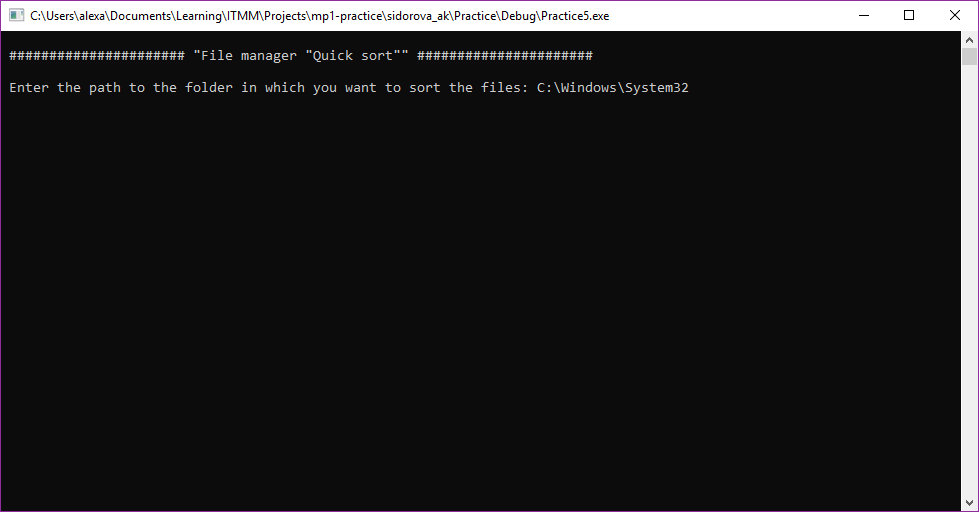


Рис. 2

После ввода отобразится список файлов, которые хранятся в данном каталоге, а так же их размеры в байтах (Рис. 3). Кроме того, после списка появится сообщение, в котором содержится список предложенных сортировок и которое просит ввести номер той сортировки, с помощью которой необходимо отсортировать файлы по их размерам. В случае, если необходимо прекратить работу с данным приложением, необходимо ввести цифру «*0*».

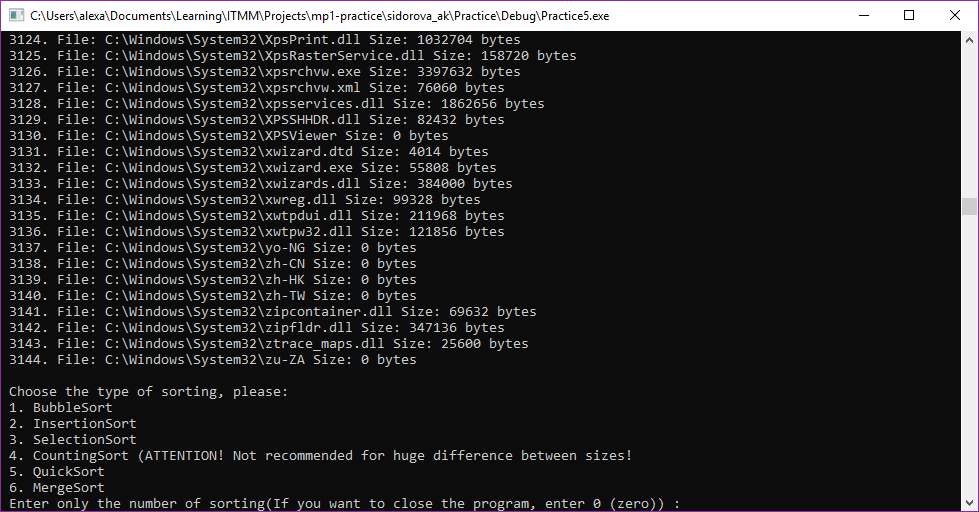


Рис. 3

Проведем для примера сортировку вставками. Для этого введем цифру «*2*» и нажмем на Enter, после чего появится новое сообщение (Рис. 4), которое предлагает два способа отсортировать: по возрастанию или по убыванию. То есть, чтобы отсортировать по возрастанию, необходимо ввести цифру «*1*», в противном случае цифру «*2*». Пусть нам необходимо отсортировать по убыванию, тогда введем цифру «*2*» и нажмем на Enter.

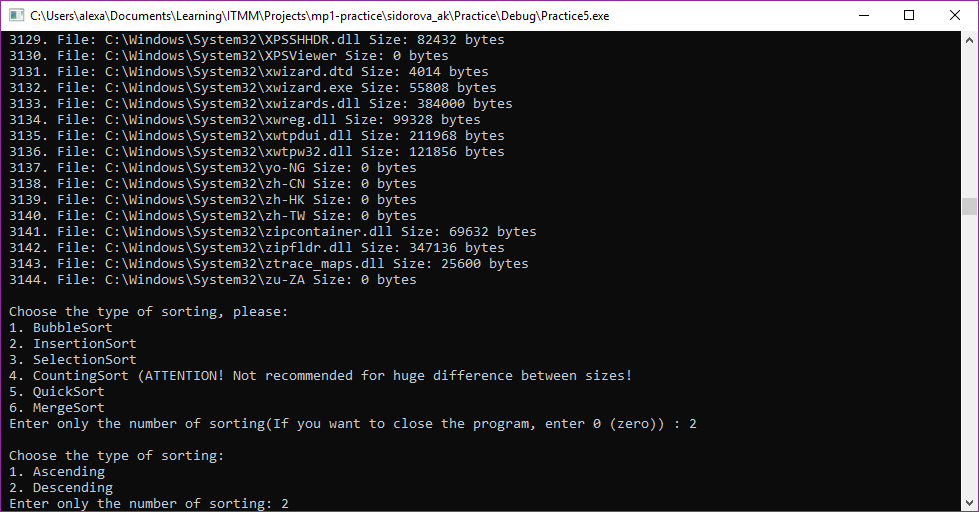


Рис. 4

Таким образом, на выходе мы получаем отсортированный по убыванию список файлов по их размерам с помощью сортировки вставками (Рис. 5). Кроме того, после списка программа выводит время, которое было затрачено на сортировку. В нашем примере было затрачено 0.058 секунд.

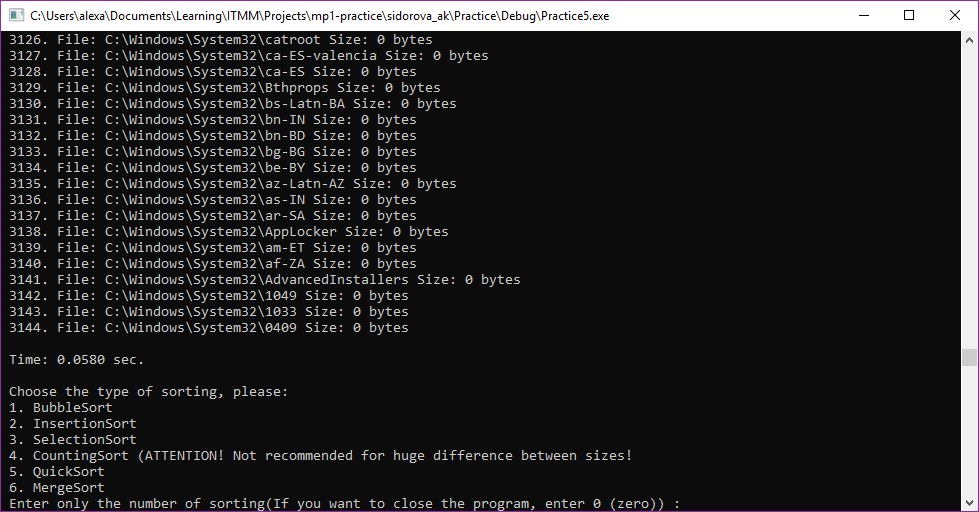


Рис. 5

Теперь пользователь может снова попробовать отсортировать файлы в заданном каталоге другой сортировкой и другим способом. Для этого необходимо повторить все те же действия, что были после ввода пути до каталога.

Как было сказано ранее, если необходимо выйти из программы, требуется ввести цифру «*0*» при выводе сообщения с видами сортировок.

# Руководство программиста

## Структура программы

Программа состоит из одного файла main.c, в котором описан весь ее код.

## Описание алгоритмов

Алгоритмы сортировок:

1. BubbleSort - «Пузырьковая» сортировка

Осуществляется несколько просмотров массива, а именно сначала от последнего до первого элемента, потом от последнего до второго, затем от последнего до третьего и т.д.

При просмотрах сравниваются соседние элементы. Если они не упорядочены относительно друг друга, то меняются местами. В результате каждого просмотра в начало текущего подмассива всплывает наименьший элемент (или наибольший, в зависимости от выбранного способа сортировки).

Рассмотрим на примере.

Пусть необходимо отсортировать массив {4, 9, 7, 6, 2, 3} по возрастанию. Расположим сверху вниз, то есть от нулевого элемента {4} до последнего {3} (рис. 6), и начнем сначала просматривать элементы снизу вверх от самого последнего до нулевого элемента. Поскольку предпоследний элемент {2} не больше {3}, менять местами их не нужно. А вот третий элемент {6} больше четвертого {2}, следовательно, их нужно поменяться местами (второй столбик на Рис. 6). Таким образом пройдемся до нулевого элемента и на данном нулевом проходе нулевым элементом станет наименьший элемент из массива.

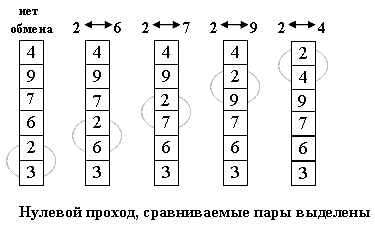


Рис. 6

Аналогично необходимо пройтись до четвертого прохода, когда массив уже будет отсортирован, поскольку предпоследний элемент {7} меньше {9}. Если бы пример был взят другой, то было бы возможно, что случай на последнем проходе был бы другим, тогда бы пришлось поменять местами два последних элемента.

Таким образом, массив отсортирован.

Реализация:

for (i = 0; i < N; i++) // До какого элемента будет выполнять просмотр

for (j = N - 1; j > i; j--) // Между какими элементами

{

if (filesSize[j - 1] > filesSize[j])

{

// Обмен элементами (размерами файлов)

Swap\_ULONGLONG(&filesSize[j - 1], &filesSize[j]);

// Обмен индексами файлов

Swap\_Ulong(&filesIndex[j - 1], &filesIndex[j]);

}

}

2. InsertionSort - Сортировка вставками

В сортировке вставками последовательно обрабатываются отрезки массива, начиная с первого элемента. Затем подмассив увеличивается, т.к. на место встаёт очередной неотсортированный элемент.

Для вставки используется буферная область памяти, в которой хранится элемент, ещё не вставленный на своё место (так называемый ключевой элемент). В подмассиве, начиная с его конца, перебираются элементы, которые сравниваются с ключевым. Если эти элементы больше ключевого, то они сдвигаются на одну позицию вправо. Если в результате этого перебора попадается элемент, меньший или равный ключевому, то значит в текущую свободную ячейку можно вставить ключевой элемент.

Рассмотрим данную сортировку на примере одного прохода.

Пусть уже отсортирована по возрастанию часть массива, например, до третьего элемента (Рис. 7).

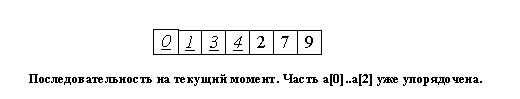


Рис. 7

Теперь у нас ключевой элемент четвертый, то есть {2}, и именно его мы сдвигаем влево и сверяемся с другими элементами, что идут до него. Так третий элемент {4} больше ключевого, поэтому мы их меняем местами (Рис. 8).

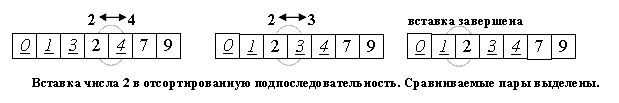


Рис. 8

Аналогично сверяем ключевой элемент со вторым элементом. Поскольку {3} больше {2}, меняем их местами. Однако первый элемент {1} уже меньше {2}, а значит, на этом шаге прекращаем этот проход и начинаем новый, в котором ключевой элемент уже пятый. Дальнейшие действия аналогичны.

Реализация:

for (i = 1; i < N; i++) // Выбор ключевого элемента

{

tmp = filesSize[i]; //Запоминаем ключевой элемент (размер)

tmpIndex = filesIndex[i]; //Запоминаем ключевой индекс файла

j = i - 1;

while ((j >= 0) && (filesSize[j] > tmp)) // Сравнения элементов

{

filesSize[j + 1] = filesSize[j]; // Сдвиг элементов

filesIndex[j + 1] = filesIndex[j]; // Сдвиг индексов файлов

j--;

}

filesSize[j + 1] = tmp; // Запись в свободную ячейку

filesIndex[j + 1] = tmpIndex;

}

3. SelectionSort – Сортировка выбором

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

На нулевом проходе идем по массиву и находим минимальный элемент. Найденный минимум меняем местами с нулевым элементом. После этого работаем с неотсортированным подмассивом, то есть от первого элемента до последнего. Там находим снова минимум, который меняем с первым элементом. И так продолжаем до тех пор, пока массив полностью не отсортируется.

Рассмотрим пример.

Предположим, что необходимо отсортировать по возрастанию данной сортировкой массив {4, 9, 7, 6, 2, 3}. На нулевом проходе найдем минимум из всех элементов массивов. Это будет четвертый элемент {2}. Его мы поменяем местами с нулевым элементом (Рис. 9). Теперь у нас не отсортирована лишь часть массива, что находится от первого элемента до последнего. Снова найдем минимум из этого подмассива. Им будет являться последний элемент {3}. Его мы поставим на место первого элемента. А первый элемент {9} поставим на последнее место (рис. 9).

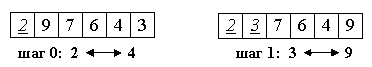


Рис. 9

Выполним аналогично еще три шага (Рис. 10).

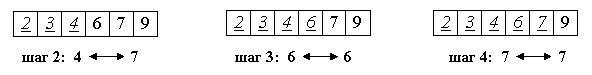


Рис. 10

Таким образом, массив отсортирован.

Причем в данном примере получилось так, что на последних трех шагах минимальные элементы стояли в начале так называемого неотсортированного подмассива. Однако далеко не всегда так получается.

Реализация:

for (i = 0; i < N; i++) // Выбираем начало неотсортированного подмассива

{

key = filesSize[i]; // Пусть минимум нулевой элемент подмассива

keyIndex = i; // Запомним его индекс

keyNewIndex = filesIndex[i]; // Запомним также индекс файла

for (j = i + 1; j < N; j++) // Пробегаем все остальные элементы

{

if (filesSize[j] < key)

{

key = filesSize[j];

keyIndex = j;

}

}

// Меняем местами элементы (размеры файлов)

Swap\_ULONGLONG(&filesSize[keyIndex], &filesSize[i]);

// Меняем местами индексы файлов, чьи размеры мы поменяли меняли

Swap\_Ulong(&filesIndex[i], &filesIndex[keyIndex]);

}

4. CountingSort – Сортировка подсчетом

Для того чтобы отсортировать массив данной сортировкой, необходим дополнительный массив. Его размер определяет разница между максимальным и минимальным элементами в неотсортированном массиве.

Чтобы отсортировать массив, необходимо пройтись от нулевого элемента до последнего. Причем каждому элементу соответствует индекс из дополнительного массива. Проходя по предложенному массиву, мы увеличиваем на единицу элемент буферного массива, индекс которого равен самому элементу, по которому мы прошлись (Или сам элемент минус минимальный). Дойдя до конца, мы начинаем обновлять наш массив. Для этого мы проходим по дополнительному массиву от начала до конца и присваиваем первоначальному нужному массиву индексы буферного, причем столько раз, сколько хранится в ячейке под таким индексом.

На выходе получаем отсортированный массив.

Рассмотрим пример.

Допустим, необходимо отсортировать данным способом по возрастанию массив {2, 5, 3, 0, 2, 3, 0, 3} (Рис. 11).

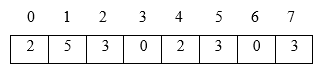


Рис. 11

Поскольку максимальный элемент {5}, а минимальный {0}, то потребуется дополнительный массив из 6 ячеек.

Пробегаем по массиву от нулевого до последнего элемента. Нулевой элемент равен {2}. Значит, добавляем единицу в элемент массива tmp с индексом 2. Теперь в ячейке буферного массива с индексом 2 хранится единица. Следующий элемент – {5}. Добавляем единицу в ячейку буферного массива с индексом 5. И так далее до тех пор, пока не дойдем до конца и не заполним дополнительный массив (Рис. 12).

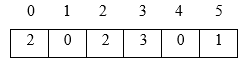


Рис. 12

Теперь необходимо с помощью дополнительного массива заполнить заново первоначально массив. Идем от начала до конца буферного массива. Нулевой элемент {2}. Следовательно, первые два элемента будут нулями. Первый элемент {0}, то есть единиц в массиве нет и идем к следующему элементу {2}. Добавим к массиву еще две двойки, поскольку индексу 2 соответствует число {2}. И так далее, пока не дойдем до конца буферного массива (Рис. 13).

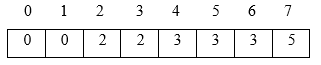


Рис. 13

Реализация:

for (i = 0; i < N; i++) // Ищем максимум и минимум

{

if (filesSize[i] < min)

min = filesSize[i];

if (filesSize[i] > max)

max = filesSize[i];

}

for (i = 0; i < N; i++) // Ищем наибольшее количество повтор. элементов

{

repeatCount = 0;

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

{

if (filesSize[i] == filesSize[j])

repeatCount++;

}

if (repeatCount > maxRepeatCount)

maxRepeatCount = repeatCount;

}

diff = max - min + 1;

// Выделяем память под буфер count

count = (ULONGLONG\*\*)malloc(diff \* sizeof(ULONGLONG\*));

/\* Делаем буфер многомерным, чтобы в остальные элементы то го же индекса записать сами индексы файлов, размеры которых и соответствуют данным индексам \*/

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

count[i] = (ULONGLONG\*)malloc(maxRepeatCount \* sizeof(ULONGLONG));

for (i = 0; i < diff; i++) // Обнуляем буфер

for (j = 0; j < MIN\_REPEAT\_FILES; j++)

count[i][j] = 0;

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

{

count[filesSize[i] - min][0]++; // Увеличиваем количество

count[filesSize[i] - min][(int)count[filesSize[i] - min][0]] = filesIndex[i]; // Записываем индексы файлов

}

for (i = 0; i < diff; i++) // Перезаписываем заново массив

for (j = 0; j < (count[i][0] + 1); j++)

if (j!= 0) filesIndex[k++] = count[i][j];

free(count); // Очищаем память

5. QuickSort – Быстрая сортировка (Сортировка Хоара)

Выбираем ключевой элемент, который будет центральным, в массиве. Разделяем массив на две части так, чтобы все элементы из левой части, которые больше или равны ключевому, перемещаем в правую, аналогично, все элементы из правой, которые меньше или равны опорному перемещаем в левую часть.

Таким образом, в левой части массива останутся элементы, которые меньше или равны центральному, а в правой — больше либо равны. Затем рекурсивно повторяем это действие для левой и правой частей массива. И так далее.

Рассмотрим пример.

Пусть нам дан массив (Рис. 14).

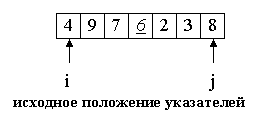


Рис. 14

Рассмотрим первое разложение. Пусть третий элемент {6} будет ключевым. Тогда массив будет разделен на две части: [0, 1, 2] и [4, 5, 6]. Будем двигаться сначала по левой. Поскольку нулевой элемент не меньше ключевого, переходим к первому и на нем останавливаемся, ведь {9} больше {6}. Запомним его индекс.

Теперь аналогично пройдемся по правой части. Т.к. предпоследний элемент {3} меньше ключевого запомним тоже его индекс.

Таким образом, мы имеем два индекса, по которым произведем обмен элементов (Рис. 15).

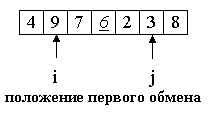


Рис. 15

Аналогично произведем второй обмен элементами и завершим проходы на данном этапе (Рис. 16).



Рис. 16

Теперь каждую часть снова поделим на два, определим ключевые в обеих частях элементы и аналогично по ним пройдемся. Таким образом, отсортируем необходимый нам массив.

Реализация функции, которая отвечает за проходы:

do

{

/\* Доходим до элемента, который больше или равен ключевому (mid), в левой части \*/

while (filesSize[\*i] < mid)

(\*i)++;

/\* Доходим до элемента, который меньше или равен ключевому (mid), в правой части \*/

while (filesSize[\*j] > mid)

(\*j)--;

if (\*i <= \*j)

{

/\* Производим обмен между элементами, отвечающим за размеры файлов, и между их индексами \*/

Swap\_ULONGLONG(&(filesSize[\*i]), &(filesSize[\*j]));

Swap\_Ulong(&(filesIndex[\*i]), &(filesIndex[\*j]));

(\*i)++;

(\*j)--;

}

} while (\*i <= \*j);

6. MergeSort – Сортировка слиянием

В данной сортировке массив разбивается на две части примерно одинакового размера. Затем каждая из получившихся частей сортируется отдельно тем же самым алгоритмом. Наконец, два упорядоченных массива половинного размера соединяются в один.

Причем рекурсивное разбиение задачи на меньшие происходит до тех пор, пока размер массива не достигнет единицы, поскольку любой массив длины 1 можно считать упорядоченным.

Соединение двух упорядоченных массивов (например, по возрастанию) в один происходит следующим образом:

Пока счетчик номеров элементов в первом подмассиве меньше индекса середины и элемент с индексом равным счетчику меньше элемента из другого подмассива или если счетчик второго подмассива перевалил за количество элементов (то есть все элементы из второго массива записаны в результирующий массив), то в результирующий массив записывается элемент из первого подмассива, а счетчик в нем увеличивается на единицу. В противном случае же записывается элемент из второго подмассива, а счетчик так же увеличивается.

Рассмотрим пример.

Предположим, что нам необходимо отсортировать таким способом по возрастанию массив {3, 7, 8, 2, 4, 6, 1, 5}. Разделим его до последовательностей длины один и начнем сливать их упорядочено (Рис. 17).

Рассмотрим первую пару. Поскольку {3} меньше {7}, то эту пару мы так и записываем первыми элементами. Теперь рассмотрим вторую пару. Так как {8} больше {2}, то мы их поменяем местами. Аналогично рассмотрим третью и четвертую пары и получим массив {3, 7, 2, 8, 4, 6, 1, 5} (Рис. 17).

Теперь рассмотрим пары пар. У нас есть первые две пары: {3, 7} и {2, 8}. Упорядочено их сольем в одну четвертку. Поскольку {2} меньше {3}, то нулевым элементов в четвертке будет {2}. Теперь рассмотрим {3} и {8}: {3} меньше{8}, а значит, первым элементом в четвертке будет {3}. Остались {7} и {8}. Так как {7} меньше {8}, то запишем сначала {7}, а затем {8}. Таким образом, мы получили четвертку вида {2, 3, 7, 8}.

Аналогично рассмотрим и вторую четвертку (Рис. 17).

Осталось теперь слить упорядочены две четвертки в целый массив. Делается это аналогично слиянию пар и четверок. И таким образом, на выходе получим массив {1, 2, 3, 4, 5, 6, 7, 8} (рис. 17)

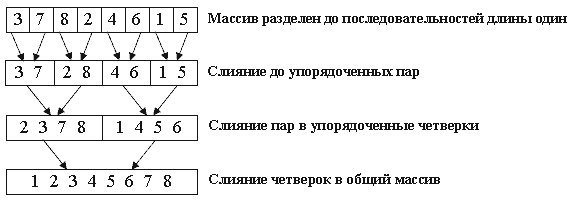


Рис. 17

Реализация:

Рассмотрим самое слияние.

// Идем по индексам результирующего массива

for (step = 0; step < last - first + 1; step++)

if ((j > last) || ((i <= midIndex) && (filesSize[i] < filesSize[j]))) // Условия записи в массив элементов из первого подмассива

{

tmp[step] = filesSize[i];

tmpIndex[step] = filesIndex[i++];

}

else

{

tmp[step] = filesSize[j];

tmpIndex[step] = filesIndex[j++];

}

// Перезапись из буфера в нужные массивы

for (step = first; step < last + 1; step++)

{

filesSize[step] = tmp[step - first];

filesIndex[step] = tmpIndex[step - first];

}

## Описание функций

void InputDirectory(wchar\_t \*\*sDir)

**Назначение:** ввод пути до каталога.

**Входные параметры:** wchar\_t \*\*sDir - указатель на массив широких символов, в который будет записан введенный пользователем путь до каталога.

**Выходные параметры:** отсутствуют.

void OutputDirectory(wchar\_t \*\*filesName, ULONGLONG \*filesSize, unsigned long \*filesIndex, unsigned long N, int ascDescType)

**Назначение:** вывод содержимого каталога в виде списка файлов и их размеров в байтах.

**Входные параметры:** wchar\_t \*\*filesName - двумерный массив широких символов, в котором хранятся имена файлов из заданного каталога, ULONGLONG \*filesSize - беззнаковый целочисленный массив, содержащий размеры в байтах файлов из заданного каталога, unsigned long \*filesIndex - беззнаковый целочисленный массив индексов файлов из заданного каталога, unsigned long N - беззнаковая целочисленная переменная, отвечающая за количество файлов в каталоге, int ascDescType - целочисленная переменная, отвечающая за способ сортировки, то есть за возрастание/убывание.

**Выходные параметры:** отсутствуют.

int ListDirectoryContents(const wchar\_t \*sDir, wchar\_t \*\*filesName, ULONGLONG \*filesSize)

**Назначение:** запись количества файлов в заданном каталоге, запись в массив имена файлов в заданном каталоге и запись в массив их размеров.

**Входные параметры:** const wchar\_t \*sDir - массив широких символов, в котором хранится путь до заданного каталога, wchar\_t \*\*filesName - двумерный массив широких символов, в котором хранятся имена файлов из заданного каталога (куда запишутся), ULONGLONG \*filesSize - беззнаковый целочисленный массив, содержащий размеры в байтах файлов из заданного каталога (куда запишутся).

**Выходные параметры:** целочисленная переменная равная количеству файлов в заданном каталоге.

void TypeSort(int \*tSort, int \*ascDescTSort)

**Назначение:** вывод меню выбора типа сортировки и способа сортировки, а также выхода из программы.

**Входные параметры:** int \*tSort - указатель на целочисленную переменную, отвечающую за тип сортировки и принимающую значение от нуля до шести (0 – выход из программы), int \*ascDescTSort - указатель на целочисленную переменную, отвечающую за способ сортировки и принимающую значение либо единицу, либо двойку.

**Выходные параметры:** отсутствуют.

void Swap\_Ulong(unsigned long \*a, unsigned long \*b)

**Назначение:** обмен элементов в массиве типа unsigned long.

**Входные параметры:** unsigned long \*a, unsigned long \*b – указатели на беззнаковые целочисленные переменные, которые являются элементами массива и которые необходимо поменять местами.

**Выходные параметры:** отсутствуют.

void Swap\_ULONGLONG(ULONGLONG \*a, ULONGLONG \*b)

**Назначение:** обмен элементов в массиве типа ULONGLONG.

**Входные параметры:** ULONGLONG \*a, ULONGLONG \*b – указатели на беззнаковые целочисленные переменные, которые являются элементами массива и которые необходимо поменять местами.

**Выходные параметры:** отсутствуют.

void QuickSplit(ULONGLONG \*filesSize, unsigned long \*i, unsigned long \*j, ULONGLONG mid, unsigned long \*filesIndex)

**Назначение:** сортировка с помощью быстрой сортировки части массива (вспомогательная функция для функции QuickSort).

**Входные параметры:** ULONGLONG \*filesSize – беззнаковый целочисленный массив размеров файлов в заданном каталоге, unsigned long \*i, unsigned long \*j – указатели на беззнаковые целочисленные переменные, отвечающие за индексы крайних элементов подмассива, ULONGLONG mid – беззнаковая целочисленная переменная равная ключевому элементу, unsigned long \*filesIndex – беззнаковый целочисленный массив индексов файлов в заданном каталоге.

**Выходные параметры:** отсутствуют.

void Merge(ULONGLONG \*filesSize, unsigned long \*filesIndex, unsigned long first, unsigned long midIndex, unsigned long last)

**Назначение:** упорядоченное сливание элементов подмассивов (вспомогательная функция для функции MergeSort).

**Входные параметры:** ULONGLONG \*filesSize – беззнаковый целочисленный массив размеров файлов в заданном каталоге, unsigned long \*filesIndex – беззнаковый целочисленный массив индексов файлов в заданном каталоге, unsigned long first, unsigned long last – беззнаковые целочисленные переменные равные началу и концу подмассивов, unsigned long midIndex - беззнаковая целочисленная переменная равная серединному элементу.

**Выходные параметры:** отсутствуют.

void BubbleSort(ULONGLONG \*filesSize, unsigned long \*filesIndex, unsigned long N)

**Назначение:** сортировка массива с помощью «пузырьковой» сортировки.

**Входные параметры:** ULONGLONG \*filesSize – беззнаковый целочисленный массив размеров файлов в заданном каталоге, unsigned long \*filesIndex – беззнаковый целочисленный массив индексов файлов в заданном каталоге, unsigned long N - беззнаковая целочисленная переменная, отвечающая за количество файлов в каталоге.

**Выходные параметры**: отсутствуют.

void InsertionSort(ULONGLONG \*filesSize, unsigned long \*filesIndex, unsigned long N)

**Назначение:** сортировка массива с помощью сортировки вставками.

**Входные параметры:** ULONGLONG \*filesSize – беззнаковый целочисленный массив размеров файлов в заданном каталоге, unsigned long \*filesIndex – беззнаковый целочисленный массив индексов файлов в заданном каталоге, unsigned long N - беззнаковая целочисленная переменная, отвечающая за количество файлов в каталоге.

**Выходные параметры**: отсутствуют.

void SelectionSort(ULONGLONG \*filesSize, unsigned long \*filesIndex, unsigned long N)

**Назначение:** сортировка массива с помощью сортировки выбором.

**Входные параметры:** ULONGLONG \*filesSize – беззнаковый целочисленный массив размеров файлов в заданном каталоге, unsigned long \*filesIndex – беззнаковый целочисленный массив индексов файлов в заданном каталоге, unsigned long N - беззнаковая целочисленная переменная, отвечающая за количество файлов в каталоге.

**Выходные параметры**: отсутствуют.

void CountingSort(ULONGLONG \*filesSize, unsigned long \*filesIndex, unsigned long N)

**Назначение:** сортировка массива с помощью сортировки выбором.

**Входные параметры:** ULONGLONG \*filesSize – беззнаковый целочисленный массив размеров файлов в заданном каталоге, unsigned long \*filesIndex – беззнаковый целочисленный массив индексов файлов в заданном каталоге, unsigned long N - беззнаковая целочисленная переменная, отвечающая за количество файлов в каталоге.

**Выходные параметры**: отсутствуют.

void QuickSort(ULONGLONG \*filesSize, unsigned long \*filesIndex, unsigned long first, unsigned long last)

**Назначение:** дробление массива на подмассивы и вызов вспомогательной функции QuickSplit, чтобы отсортировать подмассивы.

**Входные параметры:** ULONGLONG \*filesSize – беззнаковый целочисленный массив размеров файлов в заданном каталоге, unsigned long \*filesIndex – беззнаковый целочисленный массив индексов файлов в заданном каталоге, unsigned long i, unsigned long j –беззнаковые целочисленные переменные, отвечающие за индексы крайних элементов подмассива.

**Выходные параметры**: отсутствуют.

void MergeSort(ULONGLONG \*filesSize, unsigned long \*filesIndex, unsigned long first, unsigned long last)

**Назначение:** дробление массива на подмассивы и вызов вспомогательной функции Merge, чтобы упорядоченно слить подмассивы в массив.

**Входные параметры:** ULONGLONG \*filesSize – беззнаковый целочисленный массив размеров файлов в заданном каталоге, unsigned long \*filesIndex – беззнаковый целочисленный массив индексов файлов в заданном каталоге, unsigned long i, unsigned long j –беззнаковые целочисленные переменные, отвечающие за индексы крайних элементов подмассива.

**Выходные данные**: отсутствуют.

void main()

**Назначение:** главная функция вызова программы.

**Входные параметры:** отсутствуют.

**Выходные параметры:** отсутствуют.

# Заключение

Таким образом, в данной практической работе был создан прототип файлового менеджера, который может отсортировать файлы в заданном пользователем каталоге шестью разными сортировками:

1. «Пузырьковая» сортировка

2. Сортировками вставками

3. Сортировка выбором

4. Сортировка подсчетом

5. Быстрая сортировка

6. Сортировка слиянием

Причем в процессе работы пользователь может сменить тип сортировки. Кроме того, пользователь может выбрать способ сортировки, то есть по возрастанию или по убыванию, а также увидеть время, затраченное на ту или иную сортировку одних и тех же файлов в заданном каталоге.

Более того, в процессе тестирования я провела собственный эксперимент в ходе, которого получила результаты, изображенные на таблице (Таблица 1), где показано, сколько времени в секундах затрачивает каждая сортировка на обработку какого-либо количества файлов в заданном каталоге.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Количество файлов в каталоге | 15 | 100 | 407 | 1576 | 3144 |
| *«Пузырьковая» сортировка (гол.)* | 0 | 0,001 | 0,005 | 0,057 | 0,217 |
| *Сортировка вставками (оранж.)* | 0 | 0 | 0,001 | 0,017 | 0,057 |
| *Сортировка выбором (фиол.)* | 0 | 0 | 0,001 | 0,007 | 0,024 |
| *Сортировка подсчетом (жел.)* | 0,546 | 3,248 | 5,888 | - | - |
| *Быстрая сортировка (син.)* | 0 | 0 | 0 | 0,001 | 0,002 |
| *Сортировка слиянием (зел.)* | 0 | 0 | 0 | 0,003 | 0,005 |

Таблица 1

Для наглядности был построен график (Рис. 18). На нем специально взята ось ординат с максимальным значением в 6 секунд, чтобы показать, как много времени при малых количествах файлов затрачивает сортировка подсчетом.

Чтобы показать наглядность работы других сортировок, был построен другой график, где ось ординат имеет максимальное значение в 0, 25 секунд (Рис. 19).

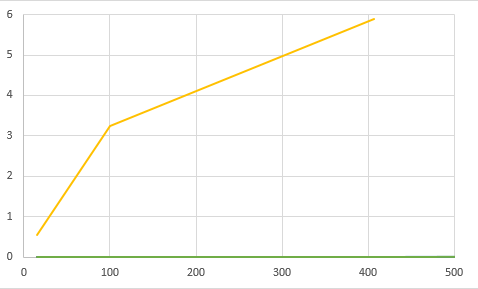


Рис. 18

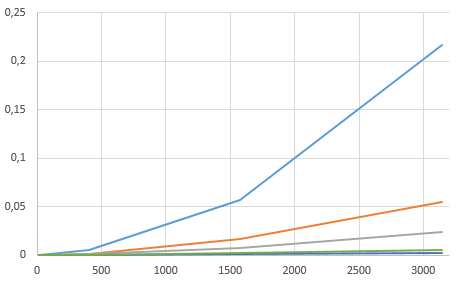


Рис. 19

Примечание: в таблице (Таблица 1) все сортировки подписаны цветом, которым они подкрашены на графиках, чтобы лучше понять, какой график к какой сортировке относится.

А также в ходе эксперимента была построена таблица средней сложности алгоритмов сортировок (Таблица 2).

|  |  |  |
| --- | --- | --- |
| Тип сортировки | Средняя сложность алгоритма | |
| **Время** | **Память** |
| *«Пузырьковая» сортировка (гол.)* | O(N2) | О(1) |
| *Сортировка вставками (оранж.)* | O(N2) | О(1) |
| *Сортировка выбором (фиол.)* | O(N2) | О(1) |
| *Сортировка подсчетом (жел.)* | О(N + (max – min)) | О((max – min)\*K\*) |
| *Быстрая сортировка (син.)* | O(N\*logN) | О(1) |
| *Сортировка слиянием (зел.)* | O(N\*logN) | О(N) |
| \* - максимальное количество повторяющихся элементов в массиве | | |

Таблица 2

Таким образом, можно сделать вывод на основе эксперимента, что из представленных сортировок самая нерациональная по памяти и времени при больших диапазонах элементов в массиве – сортировка подсчетом, а самая рациональная по памяти и времени – быстрая сортировка или сортировка Хоара.