Алгоритм – Сортировка слиянием (Merge sort).

1. Общая постановка задачи

Необходимо создать новый компонент, который должен реализовывать новый интерфейс с методом сортировки по алгоритму Сортировки слиянием (Merge sort).

Интерфейс должен содержать методы для работы с разными типами данных (int, float, double, long double).

2. Реализуемый алгоритм сортировки слиянием.

Сортировка слиянием является одним из алгоритмов Divide and Conquer – «Разделяй и властвуй». Суть таких алгоритмов заключается в разбивании текущей задачи на более простые, «решаемые» подзадачи, а затем, при помощи объединения получившихся результатов, мы можем получить ответ на нашу первоначальную задачу.

Пример: мы не знаем, как сразу отсортировать целый массив из N элементов, однако если разбить его на подмассивы из двух элементов, то чтобы отсортировать два элемента, нужно просто сравнить их между собой. Далее нужно просто объединить эти подмассивы – возьмем два соседних, оба подмассива уже отсортированы, а значит нам остается просто пройти по ним двумя указателями и по очереди сформировать объединенный массив. В этом и заключается идея Сортировки слиянием.

Задача состоит в том, чтобы отсортировать исходный массив. В ходе алгоритма сортировки Слиянием происходит следующее:

1. Массив разбивается на две части – подотрезка (условно, по середине);
2. Каждый из двух подотрезков рекурсивно разбивается на две части;
3. Разбиение происходит до тех пор, пока подотрезок не будет состоять из одного или двух элементов.
4. На каждом шаге наша функция сортирует переданный ей подотрезок. Соответственно, если элемент всего один, то подотрезок уже отсортирован, а значит мы можем его вернуть. Если же элементов два, то необходимо сравнить их между собой и поставить в порядке, заданном сортировкой. И также вернуть этот подотрезок.
5. Далее к нам приходят уже два отсортированных подотрезка, который необходимо объединить. Воспользуемся техникой «Двух указателей»: первый указатель поставим на первый элемент первого подотрезка, второй – на первый элемент второго подотрезка. Собираем итоговый массив просто сравнивая значения указателей между собой и берем либо минимальное, либо максимальное (в зависимости от сортировки). Тот указатель, чье значение мы взяли мы можем сдвинуть на следующий элемент.

В результате такого алгоритма мы получаем рекурсивно отсортированный массив. Такой алгоритм был впервые открыт в 1945 году Джоном Фон Нейманом.

3. Асимптотика.

Временная сложность:

- Средняя: O (n * log n);
- Лучшая: O (n * log n);
- Худшая: O (n * log n).

Важно пояснить, что эта сортировка является очень хорошей по временной асимптотике, так как работает за O (n * log n) даже в худшем случае. Это происходит за счет рекурсивного деления массива на 2 части – соответственно, для разбиения всего массива на части по одному – два элемента нужно сделать log n операций. На каждом таком шаге, при этом, мы на «обратном пути» соединяем массивы воедино из двух частей. То есть на каждом шаге мы проходимся по всему массиву за линию – то есть n операций.

Итоговая сложность получается: O (n * log n).

Сложность по памяти:

- O(n) – на каждом шаге при объединении двух частей мы создаем вспомогательный массив, в который складываем результат объединения. Именно такой алгоритм реализован в данной лабораторной работе.
- O(1) – существуют алгоритмы, которые позволяют сделать операцию объединения in-place (без расхода дополнительной памяти), однако в таком случае возрастает временная сложность алгоритма – из-за этого был выбран именно первый алгоритм.

4. Реализация.

Для решения задачи интерфейс содержит 4 метода:

```
typedef struct IEcoLab1VTbl {

    /* IEcoUnknown */
    int16_t (ECOCALLMETHOD *QueryInterface)(/* in */ IEcoLab1Ptr_t me, /* in */ const UGUID* riid, /* out */
voidptr_t* ppv);
    uint32_t (ECOCALLMETHOD *AddRef)(/* in */ IEcoLab1Ptr_t me);
    uint32_t (ECOCALLMETHOD *Release)(/* in */ IEcoLab1Ptr_t me);

    /* IEcoLab1 */
    int16_t (ECOCALLMETHOD *MergeSortInt)(
        /* in */ IEcoLab1Ptr_t me,
        /* in */ int* arrayPtr,
        /* in */ int size
    );

    int16_t (ECOCALLMETHOD *MergeSortLong)(
        /* in */ IEcoLab1Ptr_t me,
        /* in */ long* arrayPtr,
        /* in */ int size
    );

    int16_t (ECOCALLMETHOD *MergeSortFloat)(
        /* in */ IEcoLab1Ptr_t me,
        /* in */ float* arrayPtr,
        /* in */ int size
    );

    int16_t (ECOCALLMETHOD *MergeSortDouble)(
        /* in */ IEcoLab1Ptr_t me,
        /* in */ double* arrayPtr,
        /* in */ int size
    );

    int16_t (ECOCALLMETHOD *MergeSortLongDouble)(
        /* in */ IEcoLab1Ptr_t me,
        /* in */ long double* arrayPtr,
        /* in */ int size
    );

} IEcoLab1VTbl, *IEcoLab1VTblPtr;
```

Представленные методы «торчат наружу», основная реализация скрыта в .c файлах:

- MergeSortInt – сортировка слиянием для массивов Int;
- MergeSortLong – сортировка слиянием для массивов Long;
- MergeSortFloat – сортировка слиянием для массивов Float;

- MergeSortDouble – сортировка слиянием для массивов Double;
- MergeSortLongDouble – сортировка слиянием для массивов Long Double.

```
/*
 *
 * <сводка>
 *   Функция doMergeSortInt
 * </сводка>
 *
 * <описание>
 *   Рекурсивная функция, выполняющая сортировку слиянием, с типом Int
 * </описание>
 *
 */
static void doMergeSortInt(CEcoLab1* pCMe, int* arr, int l, int r) {
    int midIdx, firstIdx, secondIdx;
    int curPartIndex = 0;
    int* curPart = 0;
    int swapValue;

    if (l >= r) {
        return;
    } else if (l + 1 == r) {
        if (arr[l] > arr[r]) {
            swapValue = arr[l];
            arr[l] = arr[r];
            arr[r] = swapValue;
        }
        return;
    }

    midIdx = (l + r) / 2;
    doMergeSortInt(pCMe, arr, l, midIdx);
    doMergeSortInt(pCMe, arr, midIdx + 1, r);

    firstIdx = l;
    secondIdx = midIdx + 1;
    curPartIndex = 0;
    curPart = (int*)pCMe->m_pIMem->pVTbl->Alloc(pCMe->m_pIMem, sizeof(int) * (r - l + 1));

    while (firstIdx <= midIdx && secondIdx <= r) {
        if (arr[firstIdx] < arr[secondIdx]) {
            curPart[curPartIndex] = arr[firstIdx];
            curPartIndex++;
            firstIdx++;
        } else {
            curPart[curPartIndex] = arr[secondIdx];
            curPartIndex++;
            secondIdx++;
        }
    }

    while (secondIdx <= r) {
        curPart[curPartIndex] = arr[secondIdx];
        curPartIndex++;
        secondIdx++;
    }
    while (firstIdx <= midIdx) {
        curPart[curPartIndex] = arr[firstIdx];
        curPartIndex++;
        firstIdx++;
    }

    for (curPartIndex = l; curPartIndex <= r; ++curPartIndex) {
        arr[curPartIndex] = curPart[curPartIndex - l];
    }

    pCMe->m_pIMem->pVTbl->Free(pCMe->m_pIMem, curPart);
}
```

```
/*
 *
 * <сводка>
 *   Функция MergeSortInt
 * </сводка>
 *
 * <описание>
 *   Сортировка слиянием для типа Int
 * </описание>
 *
 */
static int16_t ECOCALLMETHOD CEcoLab1_MergeSortInt(
    /* in */ IEcoLab1Ptr_t me,
    /* in */ int* arrayPtr,
    /* in */ int size
) {
    CEcoLab1* pCMe = (CEcoLab1*)me;

    if (me == 0 || arrayPtr == 0 || size <= 0) {
        return ERR_ECO_POINTER;
    }
    doMergeSortInt(pCMe, arrayPtr, 0, size - 1);
    return ERR_ECO_SUCCESES;
}
```

В качестве примера: функция MergeSortInt – сортировка целых чисел. Как мы видим, в функции, объявленной в .h файле простейшая реализация – происходит проверка валидности всех пришедших данных и вызывается рекурсивная функция doMergeSortInt. В этой функции находится вся логика сортировки Слиянием.

5. Пример работы.

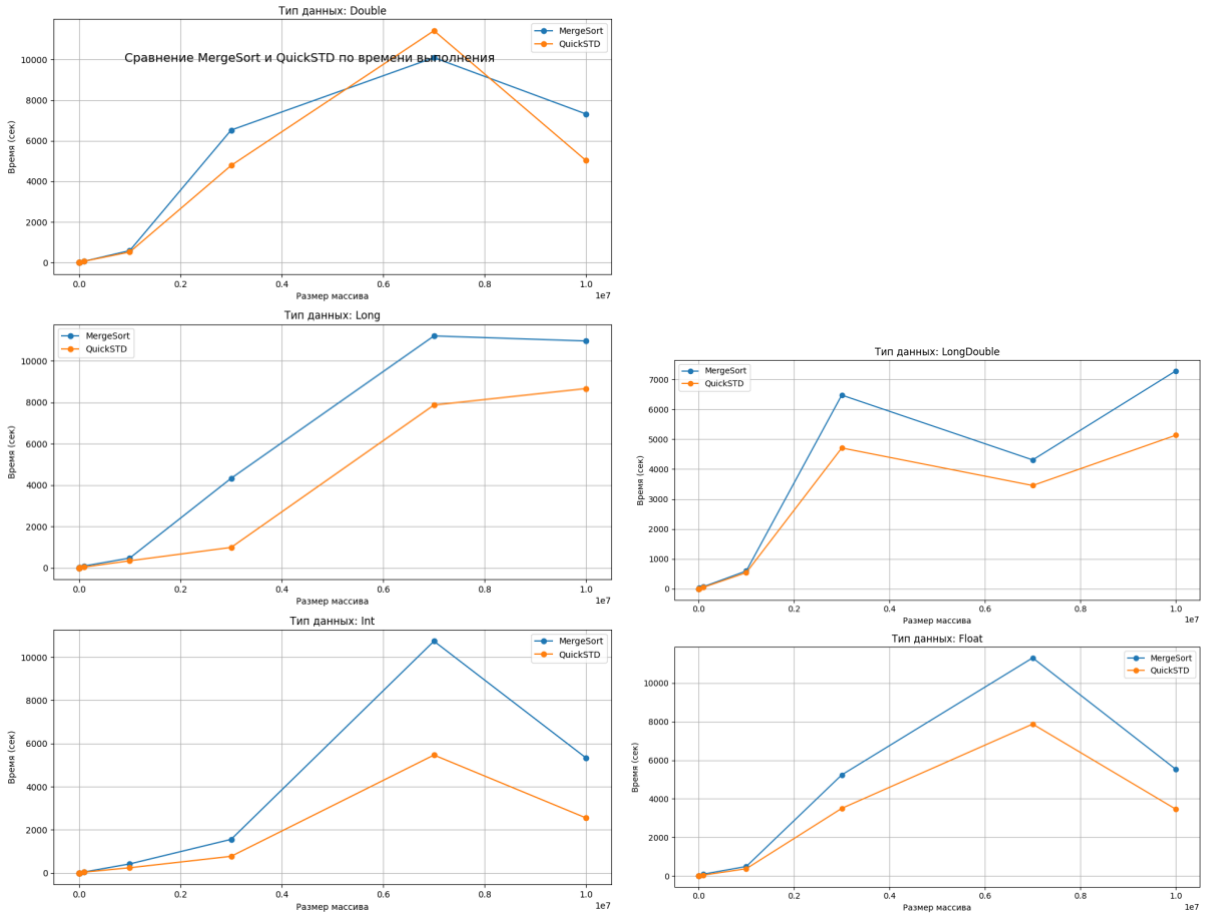
```
Random generated array: 254 -862 946 -402 198 -763 -925 -710 656 978
MergeSorted array: -925 -862 -763 -710 -402 198 254 656 946 978
Int: MergeSorted array of 10 elements in 0.000000:
```

Программа работает с массивами типов данных int, long, float, double, long double, а также с отрицательными числами.

Наша функция принимает массив чисел и возвращает его отсортированным.ы

6. Сравнение с qsort из библиотеки <stdlib.h>.

Для замера качества был написан код, замеряющий время работы нашей функции MergeSort* и функции qsort на разных массивах данных. Для удобства были построены графики зависимости времени работы от размера входного массива.



Массивы были взяты длин {10, 100, 1000, 10000, 100000, 1000000, 3000000, 7000000, 10000000}. Данные в них были случайно сгенерированы.

Результаты получились следующие: практически на всех типах данных наша функция MergeSort проигрывает qsort из стандартной библиотеки.

Можно сказать, что причинами являются:

- дополнительное время при объединении массивов – создание промежуточного массива в памяти и его последующее удаление. Скорей всего в qsort происходят in-place замены элементов за счет других видов сортировок. За счет этого не затрачивается дополнительное время;
- также важно, что существует погрешность запусков – для более корректных данных необходимо усреднять результаты по большому количеству запусков.

Можно заметить, что как таковой зависимости времени работы от типа данных не наблюдается. Так как это числа и всегда понятно, как их между собой сравнить за O(1).

7. Написание юнит тестов.

В компоненте EcoLab1 были также написаны Юнит тесты для нашего модуля. Как ранее было сказано, мы замеряли время работы по сравнению с qsort из <stdlib.h>. В юнит тестах также использовалась эта функция.

Для каждой длины массива из {10, 100, 1000, 10000, 100000, 1000000, 3000000, 7000000, 10000000} и каждого типа данных генерировался массив случайных чисел такого размера с соответствующим типом данных.

Создавалась копия нашего массива. Соответственно, один массив сортировался сортировкой qsort из <stdlib.h>, второй – нашей сортировкой MergeSort.

Для проверки корректности работы алгоритма в юнит тесте необходимо было просто проверить, что первый и второй массивы совпадают. Именно такая логика и была реализована для всех типов данных, которые поддерживает наш модуль. В случае несовпадения – выводится текст о несовпадении (вместе с не совпавшими значениями), а также несовпадение логгируется в файл stats.csv.