EcoLab1-3

Создание компонента, реализующего алгоритм сортировки подсчетом

Выполнила
Слепнева Мария
21ПИЗ

1. Алгоритм

Сортировка подсчетом - это алгоритм сортировки, который может использоваться для сортировки элементов в пределах определенного диапазона, основанный на количестве каждого сортируемого элемента в исходном массиве. При сортировке подсчетом поддерживается вспомогательный массив, который значительно увеличивает требования к памяти для реализации алгоритма. Эта сортировка базируется на целочисленных значениях исходного массива, также она не является in-place сортировкой и считается стабильной (сохраняет относительный порядок равных элементов).

Это работает следующим образом. Сначала мы вычисляем максимальное значение во входном массиве. Затем мы подсчитываем количество вхождений каждого элемента массива от 0 до длины-1 и присваиваем его вспомогательному массиву. Затем этот массив снова используется для извлечения отсортированной версии входного массива (проходимся по вспомогательному массиву и вставляем значения в исходный на основании частоты встречания элементов).

```
countingSort(arr, n)
  maximum <- find the largest element in arr
  create a count array of size maximum+1
  initialise count array with all 0's
  for i <- 0 to n
    find the total frequency/ occurrences of each element and
    store the count at ith index in count arr

  for i <- 1 to maximum
    find the cumulative sum by adding current(i) and prev(i-1) count and store
        it in count arr itself

  for j <- n down to 1
    copy the element back into the input array
    decrement count of each element copied by 1
```

Псевдокод

2. Сложность

Поскольку подсчет вхождения каждого элемента во входном диапазоне занимает k времени, а затем поиск правильного значения индекса каждого элемента в отсортированном выходном массиве занимает n времени, таким образом, общая временная сложность становится равной O(N + K).

Временная сложность в наихудшем случае

Временная сложность в наихудшем случае возникает тогда, когда данные искажены, то есть самый большой элемент значительно больше, чем другие элементы. Это увеличивает диапазон, который придется проходить, чтобы в конечном итоге обработать все числа.

Временная сложность в лучшем случае

Временная сложность в лучшем случае возникает, когда все элементы находятся в небольшом диапазоне, то есть когда массив состоит из какого-то одного повторяющегося элемента. В этом случае подсчет вхождения каждого элемента во входной диапазон занимает постоянное время, а затем поиск правильного значения индекса каждого элемента в отсортированном выходном массиве занимает n раз, таким образом, общая временная сложность уменьшается до O(1 + n), т.е. O(n), то есть сложность становится линейной.

Средняя временная сложность

Чтобы вычислить среднюю временную сложность, сначала фиксируем N и принимаем разные значения k от 1 до бесконечности, в этом случае k вычисляется как (k + 1) / 2, а средний случай будет равен N + (K + 1) / 2. Но поскольку K стремится к бесконечности, K является доминирующим фактором. Аналогично, теперь, если изменим N, то увидим, что и N, и K одинаково доминируют, и, следовательно, мы имеем O (N + K) в качестве среднего случая.

Пространственная сложность сортировки подсчетом

При реализации подсчета сортировки требуется дополнительное пространство, поскольку необходимо создать вспомогательный массив размером K + 1. Следовательно, сложность пространства равна: O(K).

Выводы

Алгоритм сортировки подсчетом - это алгоритм сортировки, не основанный на сравнении, т.е. расположение элементов в массиве не влияет на ход выполнения алгоритма. Независимо от того, отсортированы ли элементы в массиве уже, отсортированы в обратном порядке или случайным образом, алгоритм работает одинаково для всех этих случаев, и, следовательно, временная сложность для всех таких случаев одинакова, т.е. O(N + K).

Сортировка с подсчетом наиболее эффективна, если диапазон входных значений не превышает количество значений, подлежащих сортировке. В этом сценарии сложность сортировки с подсчетом намного ближе к O(N), что делает ее алгоритмом линейной сортировки.

Хоть данный алгоритм имеет линейную временную сложность, я не могу сказать, что это лучший подход, потому что пространственная сложность довольно высока, и он подходит для использования только в сценарии, где диапазон элементов входного массива близок к размеру массива. Также минусом является тот факт, что данная сортировка работает только на массивах с целочисленными типами, что делает ее не универсальной.

3. Улучшения

〉 Случай, который я описала выше, подходит только для сортировки неотрицательных чисел. Но что делать, если в исходном массиве присутствуют отрицательные числа? Для работы сортировки в этом случае необходимо сделать следующие шаги:

1. Определить минимальное и максимальное значение в исходном массиве.
2. Создать дополнительный массив для хранения количества каждого элемента от минимального до максимального значения + 1.
3. Преобразовать числа, добавив к каждому элементу значение минимального элемента (для того, чтобы индексы не были отрицательными).
4. Посчитать количество каждого элемента в исходном массиве и заполнить соответствующие ячейки дополнительного массива.
5. Восстановить отсортированный массив из дополнительного массива, помня, что необходимо снова прибавить минимальное значение к каждому элементу, чтобы вернуться к исходным числам.

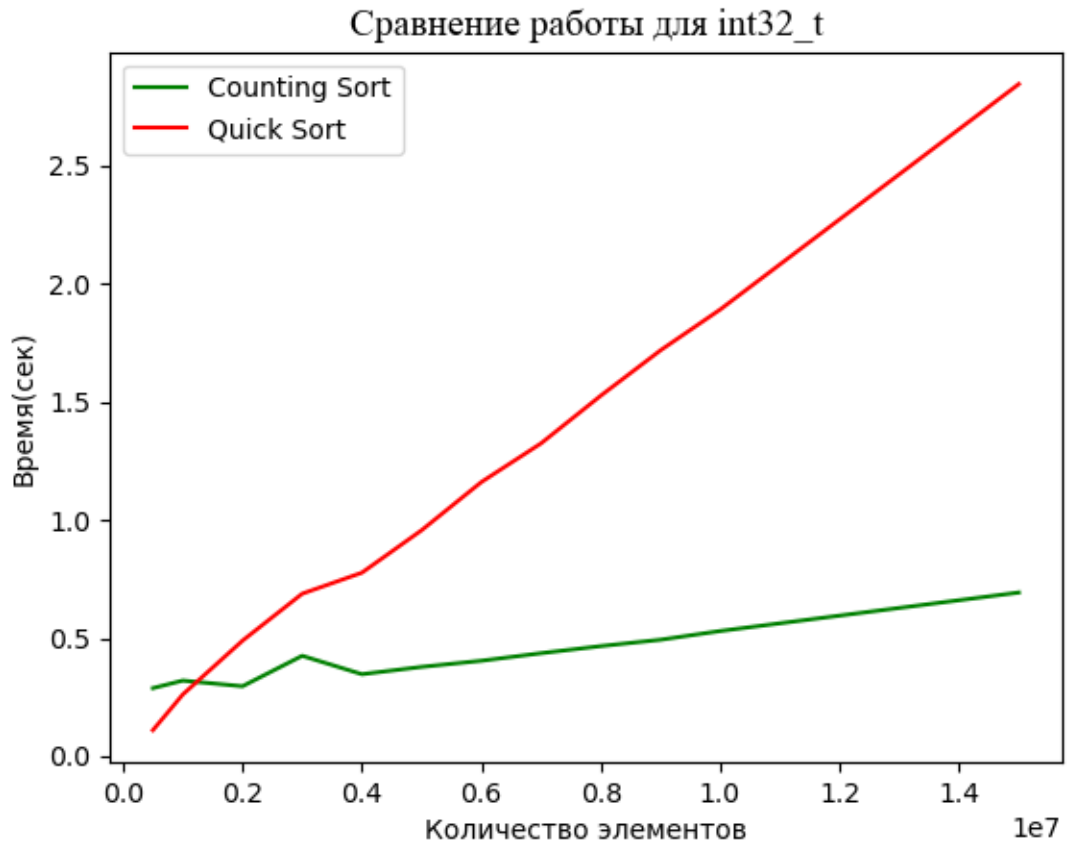
В этом случае сложность остается та же, но K здесь - это диапазон значений (наибольшее минус наименьшее число в исходном массиве).

〉 Также был разработан отдельный компонент для сортировки строки.
Компонент предоставляет пользователю сортировать строку за линейное время.

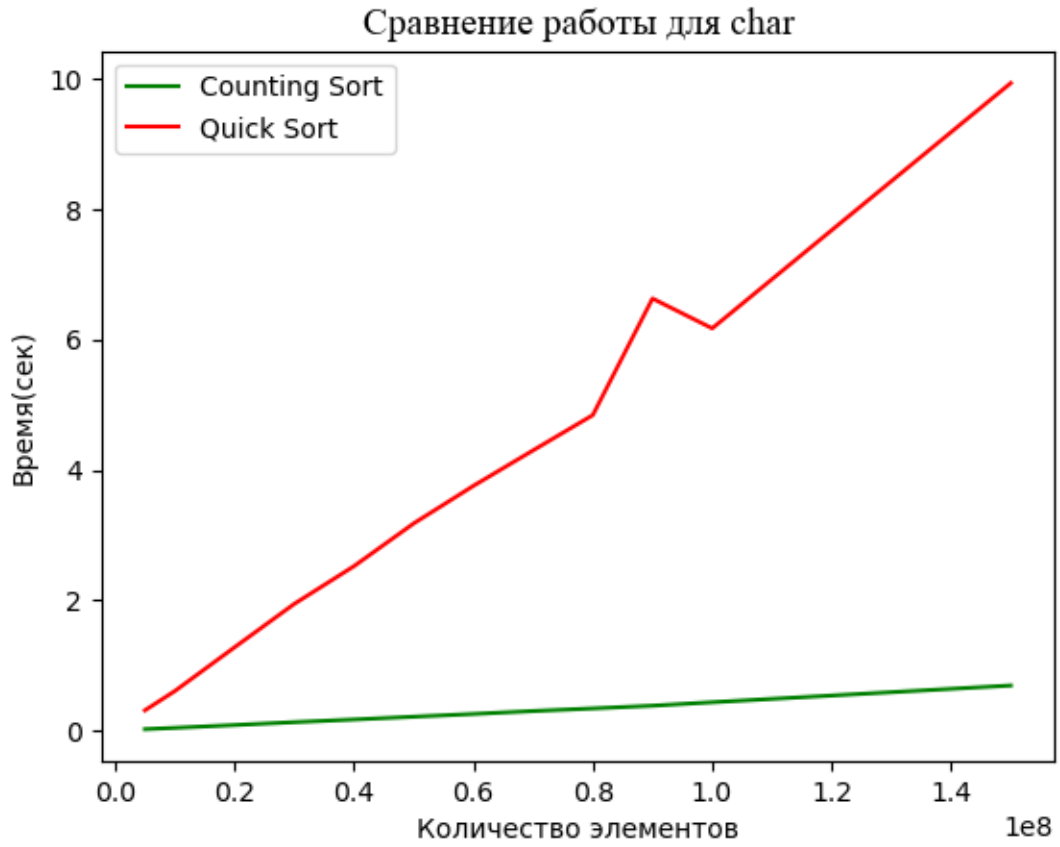
- Таким образом, в ходя работы было разработано два компонента
  - **1F5DF16EE1BF43B999A434ED38FE8F3A** - Компонент с версией сортировки подсчетом для массива целых чисел(int32_t*)
  - **1F5DF16EE1BF43B999A434ED38FE8F3B** - Компонент с версией сортировки подсчетом для строки(char*)

4. Результаты тестирования

Так как данный алгоритм подразумевает работу исключительно на целочисленных значениях, то тестирование программы было проведено именно на этом типе данных. Был рассмотрен только int32, поскольку при попытке сгенерировать доп массив для чисел типа int64 моя программа падала. Думаю, что это зависит от ЭВМ моего компьютера. Очень возможно, что на другом компьютере памяти бы хватило, но все-таки данный эксперимент доказал, что сортировка подсчетом требует больших пространственных затрат, а значит это не самый универсальный способ упорядочивания значений в массиве. Для чистоты эксперимента использовался рандомный int32. Результаты тестирования можно увидеть на графике ниже (время указано в секундах):



Также было проведено тестирование второго компонента. В данном случае сортировка подсчетом показывает лучшие результаты, так как диапазон значений символов составляет 256 элементов.



На основании этого я могу сделать следующий вывод - как видно, сортировка подсчетом действительно имеет линейную сложность. Особенно хорошо это видно в сравнении с qsort. Причем на больших данных сортировка подсчетом показала себя лучше! Но, конечно же стоит учитывать, что за быстроту работы алгоритма пришлось платить очень большими затратами памяти.

5. Поддержка других компонентов

Задачи второй лабораторной работы:

- в ранее созданный компонент используя приемы программирования включение/ агрегирование добавить компонент(-ы) калькулятора, выполняющий(-ие) операции сложения, вычитания, деления и умножения
- продемонстрировать свойства интерфейсов, показать, что через любой указатель на интерфейс можно получить указатель на любой другой запрашиваемый интерфейс используя метод QueryInterface

Для поддержки сторонних компонентов в структуру CEcoLab1 были добавлены указатели на новые виртуальные таблицы и соответствующие интерфейсы:

```
typedef struct CEcoLab1 {

    /* Таблица функций интерфейса IEcoLab1 */
    IEcoLab1VTbl* m_pVTblIEcoLab1;


    /* Счетчик ссылок */
    uint32_t m_cRef;

    /* Интерфейс для работы с памятью */
    IEcoMemoryAllocator1* m_pIMem;

    /* Системный интерфейс */
    IEcoSystem1* m_pISys;

    /* Данные экземпляра */
    char_t* m_Name;

    /* Таблица виртуальных функций интерфейса IEcoCalculatorX */
    IEcoCalculatorXVTbl* m_pVTblIEcoCalculatorX;

    /* Таблица виртуальных функций интерфейса IEcoCalculatorY */
    IEcoCalculatorYVTbl* m_pVTblIEcoCalculatorY;

    /* Указатель на интерфейс IEcoCalculatorX включаемого компонента */
    IEcoCalculatorX* m_pIEcoCalculatorX;

    /* Указатель на интерфейс IEcoCalculatorY включаемого компонента */
    IEcoCalculatorY* m_pIEcoCalculatorY;

    /* Указатель на интерфейс внутреннего агрегируемого компонента*/
    IEcoUnknown* m_pInnerUnknown;

} CEcoLab1, *CEcoLab1Ptr;
```

В результате проделанной работы были изменены следующие методы в CEcoLab1.c:

- initCEcoLab1
- deleteCEcoLab1
- CEcoLab1_QueryInterface

Также добавлены методы сложения, вычитания, умножения и деления, имплементированные с помощью внешних компонентов и включенные в таблицу виртуальных функций:

- CEcoLab1_Addition
- CEcoLab1_Subtraction
- CEcoLab1_Multiplication
- CEcoLab1_Division

Пример метода включения и взаимозаменяемости компонентов. В данном случае, если не удалось получить компонент A, получаем компонент B.

```
/* Включение IEcoCalculatorX из CEcoCalculatorA */
result = pIBus->pVTbl->QueryComponent(pIBus, &CID_EcoCalculatorA, 0, &IID_IEcoCalculatorX, (void**)&pCMe->m_pIEcoCalculatorX);
if (result != 0 || pCMe->m_pIEcoCalculatorX == 0) {
     /* В случае ошибки включение IEcoCalculatorX из CEcoCalculatorB */
    result = pIBus->pVTbl->QueryComponent(pIBus, &CID_EcoCalculatorB, 0, &IID_IEcoCalculatorX, (void**)&pCMe->m_pIEcoCalculatorX);
}
if (result == 0) {
    pCMe->m_pVTblIEcoCalculatorX = pCMe->m_pIEcoCalculatorX->pVTbl;
}
```

Когда компонент доступен, мы можем имплементировать методы интерфейса внутри нашего компонента и определить в виртуальной таблицу. Для этого добавим условие в методе CEcoLab1_QueryInterface.

```
else if (IsEqualUGUID(riid, &IID_IEcoCalculatorX)) {
    *ppv = &pCMe->m_pVTblIEcoCalculatorX;
    pCMe->m_pVTblIEcoLab1->AddRef((IEcoLab1*)pCMe);
}
```

Пример получения интерфейса методом агрегирования. В данном случае, если не удалось получить компонент E, методом включения получаем компонент D.

```
/* Агрегирование CEcoCalculatorE для доступа к IEcoCalculatorY */
result = pIBus->pVTbl->QueryComponent(pIBus, &CID_EcoCalculatorE, pOuterUnknown, &IID_IEcoUnknown, (void**)&pCMe->m_pInnerUnknown);
if (result != 0 || pCMe->m_pInnerUnknown == 0) {
    /* В случае ошибки включаем IEcoCalculatorY из CEcoCalculatorD */
    result = pIBus->pVTbl->QueryComponent(pIBus, &CID_EcoCalculatorD, 0, &IID_IEcoCalculatorY, (void**)&pCMe->m_pIEcoCalculatorY);
    if (result == 0) {
        pCMe->m_pVTblIEcoCalculatorY = pCMe->m_pIEcoCalculatorY->pVTbl;
    }
}
```

В методе CEcoLab1_QueryInterface также добавляем новое условие, которое проверяет, действительно ли пользователь запрашивает поддерживаемый компонент, а также сравнивает указатель на этот компонент (m_pInnerUnknown) с нулем. Если данный указатель не равен нулю, то вызов метода происходит по нему.

```
else if (IsEqualUGUID(riid, &IID_IEcoCalculatorY) && pCMe->m_pInnerUnknown != 0) {
    return pCMe->m_pInnerUnknown->pVTbl->QueryInterface(pCMe->m_pInnerUnknown, riid, ppv);
}
```

Тестирование

Для проверки поддержки агрегирования, включения и взаимозаменяемости были написаны запросы интерфейсов (реализация в EcoLab1.c в функции EcoMain).

```
Test interfaces:
Query: IEcoCalculatorX from EcoLab1 is correct!
Query: IEcoCalculatorY from EcoLab1 is correct!
Query: IEcoCalculatorX from IEcoCalculatorY is correct!
Query: IEcoCalculatorY from IEcoCalculatorX is correct!
```

Также там проверяется корректность работы функций, использующих имплементацию сторонних компонентов.

```
Test operations:
_____________________

Addition test
20 + 5 ==    25 - Equal!
_____________________

Subtraction test
20 - 5 ==    15 - Equal!
_____________________

Multiplication test
20 * 5 ==   100 - Equal!
_____________________

Division test
20 / 5 ==     4 - Equal!
_____________________
```

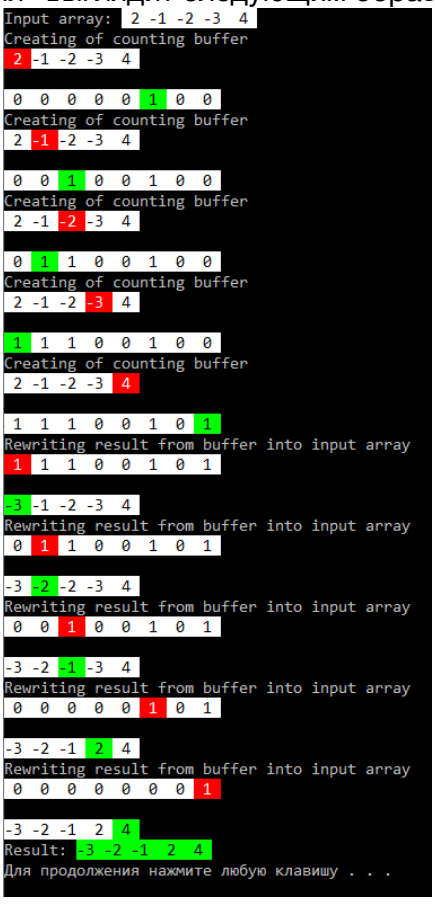
6. Визуализация

Для визуализации были имплементированы обратные вызовы, с целью демонстрации работы сортировки подсчетом.

Для более приятного восприятия использовались дополнительные цвета в консольном выводе:

- Белый
- Зеленый
- Красный

Каждый массив выделен белым в консоли.
Визуализация без «перетирания» выглядит следующим образом для 5 элементов:



Довольно масштабно для 5 элементов. Поэтому с целью лучшего восприятия и большей наглядности было принято решение не выводить все строчки, а показывать текущую итерацию алгоритма в отдельности с некоторой задержкой по времени.

Для удобства было создано 4 обратных вызова:

- BeginSort выводит массив, выделяя его белым
- IterCreateBuffSort показывает итерацию созданию буфера из исходного массива.

Первый массив – исходный, текущий элемент подсвечивается красным. Второй массив – это буфер, хранящий количество элементов по номеру индекса, соответствующего определенному значению.

- IterWriteResSort показывает итерацию записывания результата из буфера.

Первый массив – буфер, текущий элемент подсвечивается красным. Второй массив – это результирующий. В качестве результирующего используется исходный, поэтому значения в нем перетираются.

- FinishSort выводит результирующий массив, подсвечивая зеленым в качестве ответа.

```
/* IEcoLab1Events */
int16_t (ECOCALLMETHOD *BeginSort)(struct IEcoLab1Events* me, const void* data, size_t size);
int16_t (ECOCALLMETHOD *IterCreateBuffSort)(struct IEcoLab1Events* me, const void* buff, size_t buff_size, size_t buff_ind, const void* data, size_t data_size, size_t data_ind);
int16_t (ECOCALLMETHOD *IterWriteResSort)(struct IEcoLab1Events* me, const void* buff, size_t buff_size, size_t buff_ind, const void* data, size_t data_size, size_t data_ind);
int16_t (ECOCALLMETHOD *FinishSort)(struct IEcoLab1Events* me, const void* data, size_t size);
```

Демонстрацию визуализации можно посмотреть в файле demo.mp4.