EcoLab1

Создание компонента, реализующего алгоритм сортировки вставками

Выполнила
Захарова
Елизавета
21ПИЗ

1. Алгоритм

Сортировка вставками - алгоритм сортировки, в котором элементы входной последовательности просматриваются по одному, и каждый новый поступивший элемент размещается в подходящее место среди ранее упорядоченных 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 встречаемости элементов).

На вход алгоритма полается последовательность n чисел. Сортируемые числа также называют ключами. Входная последовательность на практике представляется в виде массива с n элементами. На выходе алгоритм должен вернуть перестановку исходной последовательности в отсортированном виде. В начальный момент отсортированная последовательность пуста. На каждом шаге алгоритма выбирается один из элементов входных данных и помещается на нужную позицию в уже отсортированной последовательности до тех пор, пока набор входных данных не будет исчерпан. В любой момент времени в отсортированной последовательности элементы удовлетворяют требованиям к выходным данным алгоритма.

```
for i = 2 to n do
    x = A[i]
    j = i
    while (j > 1 and A[j-1] > x) do
        A[j] = A[j-1]
        j = j - 1
    end while
    A[j] = x
end for[6]
```

Псевдокод

2. Сложность

Время выполнения алгоритма зависит от входных данных: чем большее множество нужно отсортировать, тем большее время потребуется для выполнения сортировки. Также на время выполнения влияет исходная упорядоченность массива. Время работы алгоритма для различных входных данных одинакового размера зависит от элементарных операций, или шагов, которые потребуется выполнить.

Временная сложность в наихудшем случае

Наихудшим случаем является массив, отсортированный в порядке, обратном нужному. При этом каждый новый элемент сравнивается со всеми в отсортированной последовательности. В данном случае время работы является квадратичной функцией от размера входных данных.

Временная сложность в лучшем случае

Самым благоприятным случаем является отсортированный массив. При этом все внутренние циклы состоят всего из одной итерации. Время работы линейно зависит от размера входных данных и составляет O(n).

Средняя временная сложность

Для анализа среднего случая нужно посчитать среднее число сравнений, необходимых для определения положения очередного элемента. При добавлении нового элемента потребуется, как минимум, одно сравнение, даже если этот элемент оказался в правильной позиции. i-й добавляемый элемент может занимать одно из i+1 положений. Предполагая случайные входные данные, новый элемент равновероятно может оказаться в любой позиции. Среднее число сравнений для вставки i-го элемента:

$$T_i = \frac{1}{i+1}\left(\sum_{p=1}^{i} p + i\right) = \frac{1}{i+1}\left(\frac{i(i+1)}{2} + i\right) = \frac{i}{2} + 1 - \frac{1}{i+1}$$

Для оценки среднего времени работы для n элементов нужно просуммировать все

T_i :

$$T(n) \approx \frac{n^2 - n}{4} + (n - 1) - (\ln(n) - 1) = O(n^2)$$

Временная сложность алгоритма – O(n^2). Однако из-за константных множителей и членов более низкого порядка алгоритм с более высоким порядком роста может выполняться для небольших входных данных быстрее, чем алгоритм с более низким порядком роста.

3. Улучшения

Бинарные вставки

Теперь вместо линейного поиска позиции мы будем использовать бинарный поиск, следовательно, количество сравнений изменится с квадрата на n*logn. Количество сравнений заметно уменьшилось, но для того, чтобы поставить элемент на своё место, всё ещё необходимо переместить большое количество элементов. В итоге время выполнения алгоритма асимптотически не уменьшилось. Бинарные вставки выгодно использовать только в случае, когда сравнение занимает много времени по сравнению со сдвигом. Например, когда мы используем массив длинных чисел.

Двухпутевые вставки

Суть этого метода в том, что вместо отсортированной части массива мы используем область вывода. Первый элемент помещается в середину области вывода, а место для последующих элементов освобождается путём сдвига элементов влево или вправо туда, куда выгоднее. Как можно заметить структура поля вывода имеет сходство с деком, а именно мы выбираем край, к которому ближе наш элемент, затем добавляем с этой стороны наш элемент и двигаем его. Итоговое число необходимых операций составит n^2/4+n*logn

4. Результаты тестирования

Было проведено тестирование программы с различными входными данными (рандомный int, рандомный double, рандомный float и рандомный string). Результаты тестирования можно увидеть на графиках ниже.

График сравнения int (вертикальная ось – логарифмическая, отвечает за время, выраженное в секундах, горизонтальная ось – размеры массивов).

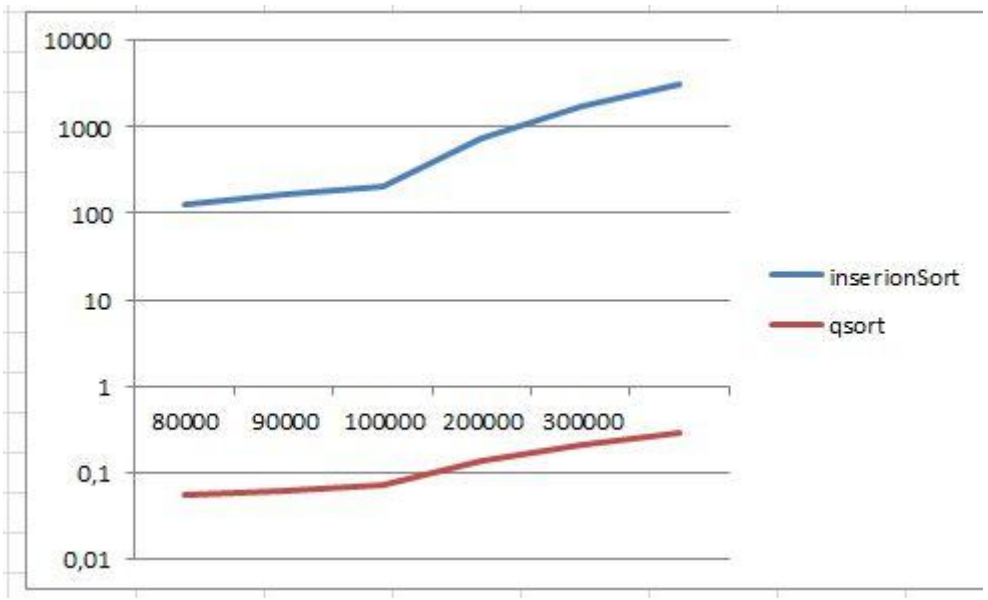


График сравнения float (вертикальная ось – логарифмическая, отвечает за время, выраженное в секундах, горизонтальная ось – размеры массивов).

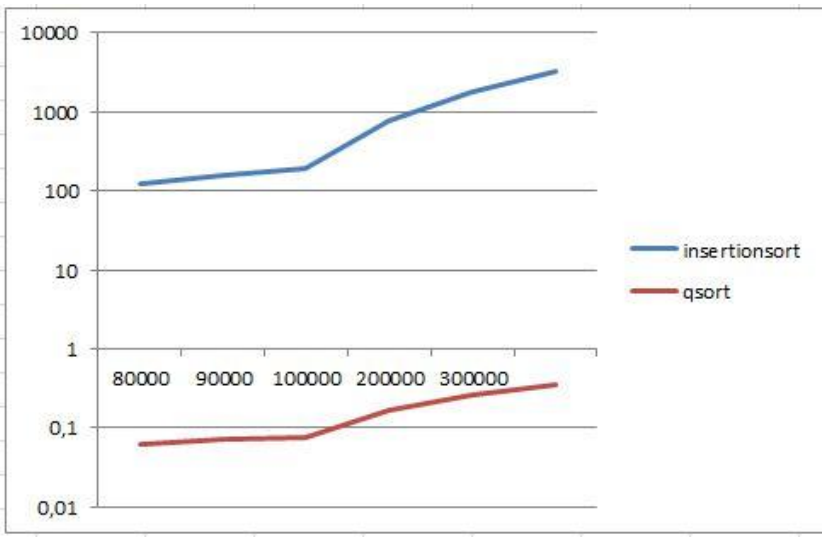


График сравнения double (вертикальная ось – логарифмическая, отвечает за время, выраженное в секундах, горизонтальная ось – размеры массивов).

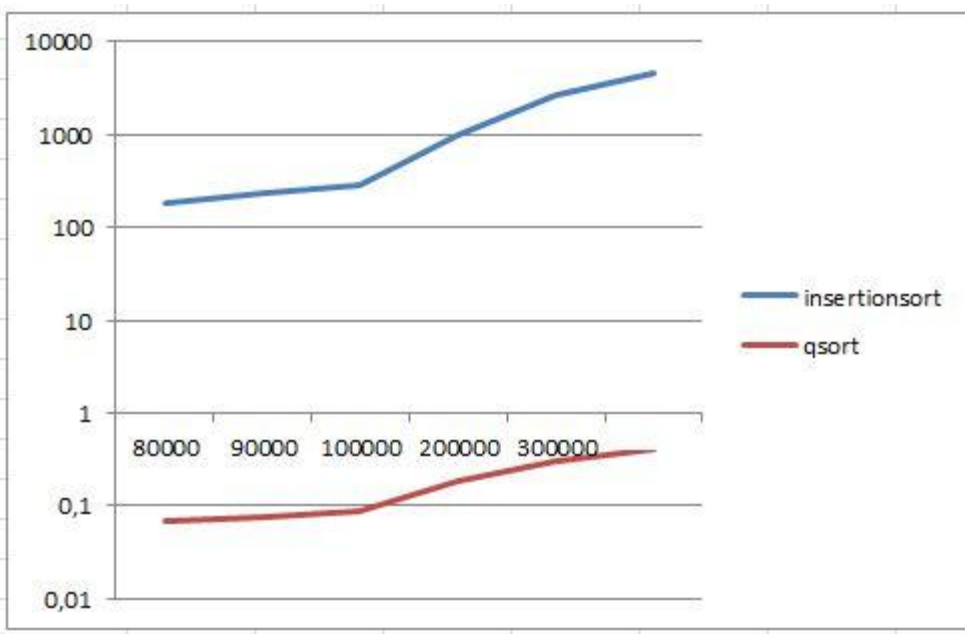
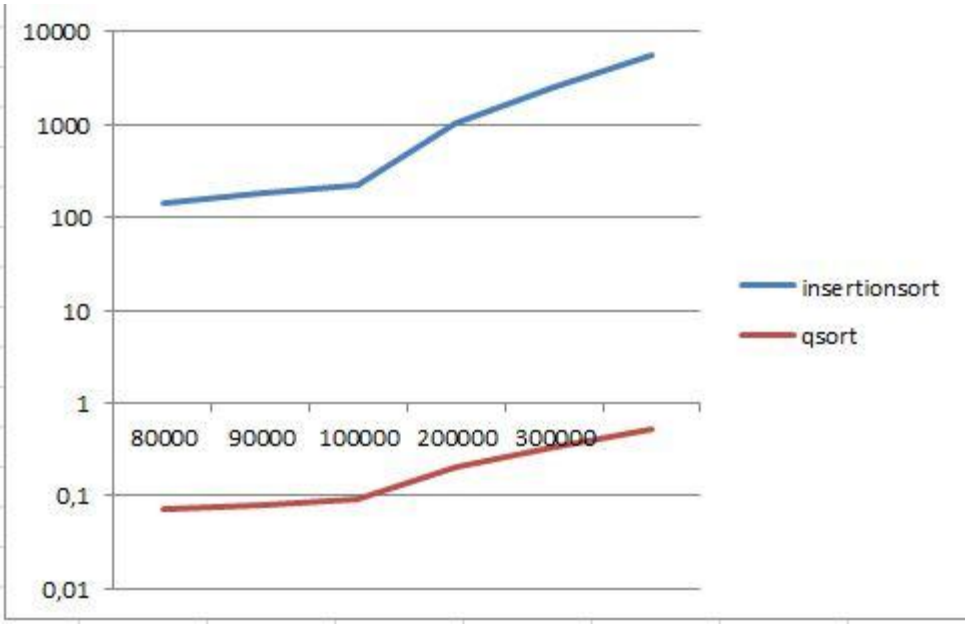


График сравнения string (вертикальная ось – логарифмическая, отвечает за время, выраженное в секундах, горизонтальная ось – размеры массивов).



Можно отметить, что время выполнения сортировки напрямую зависит от входных данных. Видно, что сортировка вставками сильно уступает qsort. Даёт о себе знать сложность O(n^2), так как приходится делать почти квадрат операций,

особенно если рандомно получается так, что массив отсортирован в обратную сторону (от большего к меньшему). А сложность qsort O(n*logn) и это видно на графиках, что даже на больших массивах она очень быстро выполняется.

5. Поддержка других компонентов

Задача 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». Демонстрационными компонентами были выбраны компоненты A,B, D и E.

Для поддержки сторонних компонентов в структуру CEcoLab1 были добавлены указатели на новые интерфейсы и их виртуальные таблицы. Для реализации метода агрегирования в структуре CEcoLab1 был создан указатель m_pInnerUnknown. Он имеет базовый для интерфейсов тип IEcoUnknown, реализующий метод QueryInterface, который нужен для передачи указателя пользователю.

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

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

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

 /* Указатели на необходимые интерфейсы */
 IEcoCalculatorX* m_pIEcoCalculatorX;

 IEcoCalculatorY* m_pIEcoCalculatorY;

 IEcoUnknown* m_pInnerUnknown;
```

Также требовалось немного изменить функции Init, QueryInterface, Delete и добавить функции, имплементирующие сложение, вычитание, умножение и деление с помощью внешних компонентов, в таблицу методов интерфейса.

Вот пример метода включения и взаимозаменяемости компонентов (получаем компонент A, если не удалось, получаем компонент B):

```
result = pIBus->pVTbl->QueryComponent(pIBus, &CID_EcoCalculatorA, 0, &IID_IEcoCalculatorX, (void**)&pCMe->m_pIEcoCalculatorX);
if (result != 0 || pCMe->m_pIEcoCalculatorX == 0) {
     // Если не получилось, то включаем IEcoCalculatorX из CEcoCalculatorA
    result = pIBus->pVTbl->QueryComponent(pIBus, &CID_EcoCalculatorB, 0, &IID_IEcoCalculatorX, (void**)&pCMe->m_pIEcoCalculatorX);

}
```

Теперь если компонент доступен, то мы можем имплементировать методы интерфейса внутри нашего компонента и определить их в виртуальную таблицу. Для передачи интерфейса пользователю добавим else if в метод CEcoLab1_QueryInterface.

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

Ниже представлен пример получения интерфейса методом агрегирования. Если интерфейс не удаётся получить из компонента E, то он будет получен из компонента D методом включения.

```
result = pIBus->pVTbl->QueryComponent(pIBus, &CID_EcoCalculatorE, pOuterUnknown, &IID_IEcoUnknown, (void**)&pCMe->m_pInnerUnknown);

if (result != 0 || pCMe->m_pInnerUnknown == 0) {
    // Если не получилось, то включаем IEcoCalculatorY из CEcoCalculatorD
    result = pIBus->pVTbl->QueryComponent(pIBus, &CID_EcoCalculatorD, 0, &IID_IEcoCalculatorY, (void**)&pCMe->m_pIEcoCalculatorY);
```

Пример запроса компонента:

Метод CEcoLab1_QueryInterface используется внутри нашего компонента с такой же модификацией, как в случае включения, то есть, else if, где, при условии, что пользователь запрашивает поддерживаемый компонент и указатель на него не равен 0, происходит вызов метода QueryInterface у полученного компонента по указателю m_pInnerUnknown.

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

Теперь компонент поддерживает агрегирование, включение и взаимозаменяемость компонентов. Юнит-тесты наглядно демонстрируют работу программы, в том числе показывают, что любой интерфейс можно получить через любой другой интерфейс