Создание компонента, реализующего алгоритм сортировки Шелла

EcoLab1

Сортировка Шелла

Выполнил

Антипов Юрий

21ПИ-2

Оглавление

[Алгоритм 2](#_Toc168245409)

[Улучшение 3](#_Toc168245410)

[Результаты тестирования 4](#_Toc168245411)

[Включение и агрегирование других компонентов 6](#_Toc168245412)

[Включение 6](#_Toc168245413)

[Агрегирование 7](#_Toc168245414)

[Визуализация 8](#_Toc168245415)

[Реализация алгоритма планирования длинная задача первая «Longest Job First» (LJF) 9](#_Toc168245416)

[Ссылки 10](#_Toc168245417)

# Алгоритм

Метод Шелла – алгоритм сортировки, не использующий дополнительную память и при этом показывающий высокую скорость. По своей сути эта сортировка применяет сравнения и перестановки элементов аналогичные методу вставок, но при этом порядок сравниваемых элементов другой. В отличие от сортировки вставками, сортировка Шелла сравнивает элементы на некотором удалении друг от друга, уменьшая это расстояние до того, как оно не станет равно 1.

Автор алгоритма Шелл предложил в качестве длин промежутков последовательность, которая начинается числом, равным половине длины сортируемого массива (естественно, приведенной к целому), и с каждым шагом уменьшается в 2 раза:

, где d – длина промежутка, N – размер сортируемого массива.

Время выполнения сортировки Шелла с такой последовательностью разрывов составляет в среднем , в худшем случае – для массива длины n.

Сортировка Шелла работает следующим образом:

1. Для массива arr размером N определяются длины промежутков;
2. Выбирается следующая длина промежутка;
3. С помощью алгоритма сортировки вставками сортируются числа на удалении выбранной длины;
4. Процесс повторяется для следующей длины промежутка.

Алгоритм завершает сортировку последним промежутком длиной 1, то есть выполняет обыкновенную сортировку вставками.

Для сортировки Шелла я изменил функцию сортировки вставками, добавив дополнительный параметр step, содержащий очередную длину промежутка, на котором и происходит сравнение элементов, в отличие от стандартной сортировки вставками, где этот параметр равен 1.

Псевдокод:

|  |
| --- |
| void ShellSort(int\* arr, int N) {  int d[];  CalculateD(d, N);  for (i = d.size() - 1; i >= 0; i--) {  InsertionSort(arr, N, d[i]);  }  } |

# Улучшение

Роберт Седжвик предложил свою формулу для последовательности длин, чтобы уменьшить число проходов по массиву с разными разрывами, при этом сохранив преимущества этого подхода:

Количество длин в таком случае сильно уменьшается, например, для массива размером 60 разбиение Шелла будет состоять из чисел [1, 3, 7, 15, 30], тогда как в реализации Седжвика понадобится только 3 числа – [1, 5, 19].

При использовании длин промежутков Седжвика средняя сложность алгоритма составляет, а в худшем случае – [[1]](#_Ссылки), что заметно лучше разбиения Шелла (см. Результаты тестирования).

# Результаты тестирования

Было проведено тестирование массивов разных типов данных и количеством элементов от 10.000 до 1.000.000. Результатом в таблице является среднее из 100 попыток время сортировки массива данной длины.

Уже из первого сравнения можно сделать вывод, насколько скорость работы сортировки Шелла зависит от разбиения. Разбиение Седжвика на всех размерах массивов справлялось быстрее разбиения Шелла, а на длине в 1.000.000 эта разница практически достигла 2-х раз. К сожалению, время работы сортировки Шелла с этим разбиением также почти в 2 раза больше, чем у встроенной qsort.

Почему-то разбиение Шелла стало работать сильно медленнее, когда тип массива поменялся на double\_t. Разница с длинами Седжвика здесь составила более 2-х раз на 1.000.000.

А вот уже отсортированный массив сортировка Шелла разбиением Седжвика “сортирует” также быстро, как встроенный qsort.

На убывающем массиве разбиение Шелла отстает от разбиения Седжвика не так сильно, как на предыдущих.

В целом, скорость работы зависит от входных данных. Лучше всего сортировка Шелла справляется с целочисленными типами и убывающей последовательностью. Разбиение Седжвика во всех случаях работало быстрее, и это при том, что время вычисления последовательности Седжвика включено во время работы всего алгоритма.

# Включение и агрегирование других компонентов

## Включение

Для поддержки других компонентов в структуру CEcoLab1 и CEcoLab2 были добавлены указатели на новые интерфейсы и их виртуальные таблицы:

|  |
| --- |
| /\* Таблица функций интерфейса IEcoCalculatorX \*/  IEcoCalculatorXVTbl\* m\_pVTblIEcoCalculatorX;  /\* Таблица функций интерфейса IEcoCalculatorY \*/  IEcoCalculatorYVTbl\* m\_pVTblIEcoCalculatorY;  …  /\* Указатель на интерфейс IEcoCalculatorX включаемого компонента \*/  IEcoCalculatorX\* m\_pIEcoCalculatorX;  /\* Указатель на интерфейс IEcoCalculatorY включаемого компонента \*/  IEcoCalculatorY\* m\_pIEcoCalculatorY; |

Пример метода включения и взаимозаменяемости компонентов D и E:

|  |
| --- |
| /\* Включение компонента Y \*/  result = pIBus->pVTbl->QueryComponent(pIBus, &CID\_EcoCalculatorD, 0, &IID\_IEcoCalculatorY, (void\*\*) &pCMe->m\_pIEcoCalculatorY);  if (result != 0 || pCMe->m\_pIEcoCalculatorY == 0) {  result = pIBus->pVTbl->QueryComponent(pIBus, &CID\_EcoCalculatorE, 0, &IID\_IEcoCalculatorY, (void\*\*) &pCMe->m\_pIEcoCalculatorY);  } |

Теперь, если компонент D или E с интерфейсом IEcoCalculatorY доступен, мы можем имплементировать методы интерфейса внутри нашего компонента и определить их в виртуальную таблицу, которая будет присвоена соответствующей переменной во время исполнения функции ***createCEcoLab1***.

Для передачи интерфейса пользователю была добавлена проверка в функцию ***CEcoLab1\_QueryInterface***:

|  |
| --- |
| else if (IsEqualUGUID(riid, &IID\_IEcoCalculatorY)) {  \*ppv = &pCMe->m\_pVTblIEcoCalculatorY;  pCMe->m\_pVTblIEcoLab1->AddRef((IEcoLab1\*) pCMe);  } |

## Агрегирование

Для реализации метода агрегирования было создано решение CEcoLab2, которое агрегирует CEcoLab1. Для этого в CEcoLab1 был добавлен указатель ***m\_pIUnkOuter*** на агрегирующий компонент:

|  |
| --- |
| /\* Делегирующий IEcoUnknown, указывает на внешний  или неделегирующий IEcoUnknown \*/  IEcoUnknown\* m\_pIUnkOuter; |

Далее, в функции ***CEcoLab1\_QueryInterface*** управление передается агрегирующему компоненту:

|  |
| --- |
| if (pCMe->m\_pIUnkOuter != 0){  return pCMe->m\_pIUnkOuter->pVTbl->QueryInterface(pCMe->m\_pIUnkOuter, riid, ppv);  } |

В свою очередь, в CEcoLab2 добавлен указатель на IEcoUnknown внутреннего компонента:

|  |
| --- |
| /\* Указатель на IEcoUnknown внутреннего компонента \*/  IEcoUnknown\* m\_pInnerUnknown; |

Он получается при инициализации CEcoLab2:

|  |
| --- |
| /\* Получение интерфейса внутреннего компонента "Eco.Lab1"  c поддержкой агрегирования \*/  /\* ВАЖНО: При агрегировании мы передаем IID IEcoUnknown \*/  result = pIBus->pVTbl->QueryComponent(pIBus, &CID\_EcoLab1, pOuterUnknown, &IID\_IEcoUnknown,(void\*\*) &pCMe->m\_pInnerUnknown); |

В конце концов, в юнит-тесте CEcoLab2 мы получаем указатель на интерфейс IEcoLab1, благодаря чему можем использовать реализуемый в нем метод ***qsort***:

|  |
| --- |
| result = pIEcoLab2->pVTbl->QueryInterface(pIEcoLab2, &IID\_IEcoLab1, (void\*\*)&pIEcoLab1);  result = pIEcoLab1->pVTbl->qsort(pIEcoLab1, size, sizeof(int32\_t), arr32, compare\_int32); |

# Визуализация

Для визуализации работы компонента было создано несколько обратных вызовов:

- OnPrintDividingType выводит тип выбранного разбиения для длины шагов сортировки вставками:

|  |
| --- |
| int16\_t (ECOCALLMETHOD \*OnPrintDividingType)(/\* in \*/ struct IEcoLab1Events\* me, /\* in \*/ int16\_t type); |

- OnNewStepSize выводит текущей размер шага для сортировки вставками:

|  |
| --- |
| int16\_t (ECOCALLMETHOD \*OnNewStepSize)(/\* in \*/ struct IEcoLab1Events\* me, /\* in \*/ uint32\_t step); |

- OnPrintIntArray выводит массив заданный массив:

|  |
| --- |
| int16\_t (ECOCALLMETHOD \*OnPrintIntArray)(/\* in \*/ struct IEcoLab1Events\* me, /\* in \*/ uint32\_t arr\_size, /\* in \*/ void\* arr[]); |

- OnPrintIntArrayBeforeSorting выводит массив с заголовком, выделенным красным цветом:

|  |
| --- |
| int16\_t (ECOCALLMETHOD \*OnPrintIntArrayBeforeSorting)(/\* in \*/ struct IEcoLab1Events\* me, /\* in \*/ uint32\_t arr\_size, /\* in \*/ void\* arr[]); |

- OnPrintIntArrayAfterSorting выводит массив с заголовком, выделенным зеленым цветом:

|  |
| --- |
| int16\_t (ECOCALLMETHOD \*OnPrintIntArrayAfterSorting)(/\* in \*/ struct IEcoLab1Events\* me, /\* in \*/ uint32\_t arr\_size, /\* in \*/ void\* arr[]); |

- OnPrintColoredIntArray выводит массив, печатая его на прошлой строке, если параметр is\_first\_print не равен 1, и выделяя заданные в idx\_to\_color элементы синим цветом:

|  |
| --- |
| int16\_t (ECOCALLMETHOD \*OnPrintColoredIntArray)(/\* in \*/ struct IEcoLab1Events\* me, /\* in \*/ uint32\_t arr\_size, /\* in \*/ void\* arr[], /\* in \*/ uint32\_t idx\_to\_color\_size, /\* in \*/ int32\_t\* idx\_to\_color[], int16\_t is\_first\_print); |

Для удобства наблюдения за визуалом callback OnPrintColoredIntArray() заканчивается функцией Sleep(), выжидающей определенное время (в данном случае 1,5 секунды) после вывода информации.

# Реализация алгоритма планирования длинная задача первая «Longest Job First» (LJF)

Для реализации алгоритма в проект Eco.TaskScheduler1Lab были внесены изменения:

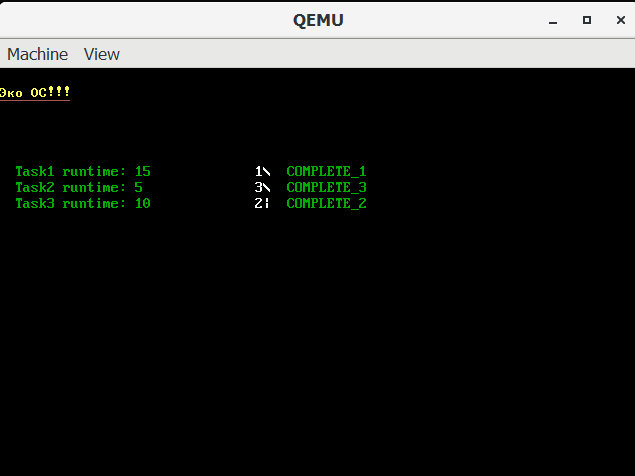
* В CEcoTask1Lab.h было добавлено поле ***runtime***, в котором хранится время выполнения данной задачи;
* При создании новой задачи в методе ***NewTask*** эта переменная инициализируется:

|  |
| --- |
| g\_xCEcoTask1List\_C761620F[indx].runtime = (uint32\_t\*)&data[0]; |
|  |

* В методе ***Start*** реализован конкретный алгоритм LIF. Задачи сортируются по убыванию времени выполнения и затем выполняются:

|  |
| --- |
| InsertionSort(pCMe, tasks, tasks\_size);  for (i = 0; i < tasks\_size; ++i) {  pCMe->m\_pTaskList[tasks[i]].pfunc();  } |
|  |

Результат выполнения программы в Qemu:



# Ссылки

1. J. Incerpi, R. Sedgewick, «Improved Upper Bounds for Shellsort», J. Computer and System Sciences 31, 2, 1985.