­­­

­­

EcoLab

Создание компонента, реализующего алгоритм

**«Stooge sort»**

Выполнил

Горячев Сергей

21ПИ-2

Оглавление

[Алгоритм 3](#_Toc168345394)

[Псевдокод 3](#_Toc168345395)

[Оценка сложности 4](#_Toc168345396)

[Общий анализ 4](#_Toc168345397)

[Выводы 4](#_Toc168345398)

[Результаты тестирования 5](#_Toc168345399)

[Случайные числа типа **int** 5](#_Toc168345400)

[Случайные числа для stooge sort 5](#_Toc168345401)

[Случайные числа для qsort 6](#_Toc168345402)

[Выводы 7](#_Toc168345403)

[Визуализация 8](#_Toc168345404)

[Реализация алгоритма планирования наиболее срочный в первую очередь «Maximum-Urgency-First» (MUF) 9](#_Toc168345405)

# Алгоритм

Stooge sort (сортировка по частям, блуждающая сортировка) – рекурсивный алгоритм сортировки с временной сложностью O(n^(log3/log1.5)) ~ O(n^2.71). �(�log1,5⁡3)≈�(�2.71)Время работы алгоритма, таким образом, крайне большое по сравнению с эффективными алгоритмами сортировки, такими, как сортировка слиянием.

Алгоритм stooge sort заключается в следующем:

1) Если значение элемента в конце списка меньше, чем значение элемента в начале, то поменять их местами.

2) Если есть 3 или более элементов в текущем подмножестве списка, то:

- Рекурсивно вызвать сортировку для первых 2/3 списка

- Рекурсивно вызвать сортировку для последних 2/3 ­­списка

- Рекурсивно вызвать сортировку для первых 2/3 списка снова

# Псевдокод

**void** sort(**int** a[], **int** lo, **int** hi) **throws** Exception {  
   
 //Сравниваем/меняем элементы на концах отрезка  
 **if**(a[lo] > a[hi]) {  
 **int** T = a[lo];  
 a[lo] = a[hi];  
 a[hi] = T;  
 }  
   
 //Меньше трёх?  
 **if**(lo + 1 >= hi) **return**;  
   
 //Чему равна одна треть?  
 **int** third = (hi - lo + 1) / 3;  
   
 sort(a, lo, hi - third); //Для первых 2/3 массива  
 sort(a, lo + third, hi); //Для последних 2/3 массива  
 sort(a, lo, hi - third); //Для первых 2/3 массива  
   
 }

# Оценка сложности

## Общий анализ

Stooge Sort - это рекурсивный алгоритм сортировки. Это не очень эффективный, но интересный алгоритм сортировки. Обычно он делит массив на две перекрывающиеся части (по 2/3 каждая). После этого он выполняет сортировку в первых 2/3 части, а затем выполняет сортировку в последних 2/3 части. И затем сортировка выполняется по первой 2/3 части, чтобы гарантировать, что массив отсортирован. Так как независимо от наличия уже упорядоченных подмассивов алгоритм никак не изменяет свои действия, то время остается одним и тем же для лучшего, среднего и худшего случая. Сложность алгоритма для всех случаев = O(n^(log(3) / log(1.5)) ~ O(n^(2.71).

## Выводы

Алгоритм stooge sort является одним из представителей самых неэффективных алгоритмов сортировки. В алгоритм заложен элемент абсурда – даже если массив давно отсортирован, сортировка продолжает безумно метаться по третям списка.

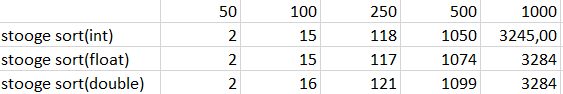
# Результаты тестирования

Было проведено тестирование алгоритма с различными входными данными: случайные числа типа **int**, случайные числа типа **float**, случайные числа типа **double**. Размеры сортируемых массивов варьировались от 10.000 элементов до 1.000.000 элементов. К сожалению, не для всех тестов удалось выполнить stooge sort, так как алгоритм работал слишком долго. Поэтому далее будут представлены результаты для qsort и stooge sort отдельно.

Результаты тестирования приведены ниже:

## Случайные числа типа **int**

## Случайные числа для stooge sort



## Случайные числа для qsort

## Выводы

Из приведенных выше данных можно сделать вывод, что время работы сортировки напрямую зависит от количества входных данных и не зависит от наличия уже упорядоченных подмассивов. В целом, моя реализация алгоритма Stooge sort сильно уступает в скорости стандартной функции сортировки qsort из stdlib. Худший результат обе функции показали на входных данных типа double, так как этот тип данных занимает наибольшее количество байтов памяти среди остальных тестируемых типов, хоть и на stooge sort это еле заметное отличие.

# Визуализация

Для визуализации добавил 2 обратных вызова:

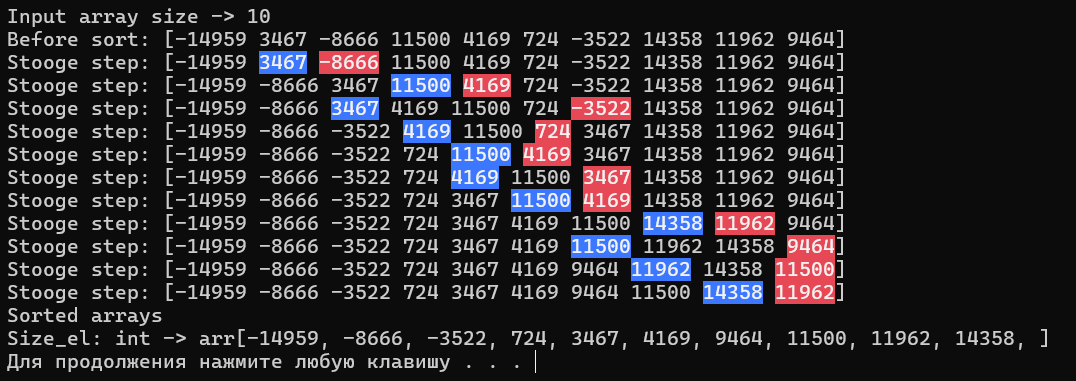
**OnStartSort** вызывается при старте работы сортировки для отображения начального состояния.

Представляет следующую структуру:

int16\_t (ECOCALLMETHOD \*OnStartSort)(struct IEcoLab1Events\* me, const void \*startPtr, size\_t elem\_count);

**OnChangeElement** вызывается во время нахождения двух элементов, которые следует поменять метсами:

int16\_t (ECOCALLMETHOD \*OnChangeElement)(struct IEcoLab1Events\* me, const void \*startPtr, size\_t elem\_count, int leftIdx, int rightIdx);

По окончанию визуализации есть время ( с помощью Sleep) ожидания перед продолжением сортировки, чтобы можно было наблюдать сортировку в реальном времени.  
  
Пример визуализации:  


# Реализация алгоритма планирования наиболее срочный в первую очередь «Maximum-Urgency-First» (MUF)

Алгоритм Maximum Urgency First (MUF) (максимальной срочности первым) используется в реальном времени для планирования задач. Он назначает процессору задачи с наивысшим приоритетом, которые должны быть выполнены в срочном порядке. Приоритет определяется на основе времени прибытия и времени выполнения задач.

Описание процесса:

typedef struct Process {

int id; // номер процесса

int arrival\_time; // время прибытия

int completion\_time; // время завершения

int burst\_time; // время действия

int priority; // приоритет

int waiting\_time; // время ожидания

} Process;

Функция сортировки по приоритету:

void ChangeElProcess(struct Process processes[], int first, int second) {

struct Process temp;

temp = processes[first];

processes[first] = processes[second];

processes[second] = temp;

}

void sortProcesses(struct Process processes[], int n) {

int count, indx;

struct Process temp;

for (count = 0; count < n - 1; count++) {

for (indx = 0; indx < n - 1 - count; indx++) {

if (processes[indx].priority < processes[indx + 1].priority) {

ChangeElProcess(processes, indx, indx+1);

}

}

}

}

Алгоритм планирования:  
  
int16\_t ECOCALLMETHOD CEcoLab1\_schProcesses(struct IEcoLab1\* me, struct Process processes[], int n, int \*completion\_order) {

CEcoLab1\* pCMe = (CEcoLab1\*)me;

int time = 0;

int completed = 0;

int is\_completed[MAX];

int i, t, j;

int index = 0;

int process\_found = 0;

for (i = 0; i < n; i++) {

is\_completed[i] = 0;

}

while (completed < n) {

process\_found = 0;

sortProcesses(processes, n);

for (i = 0; i < n; i++) {

if (processes[i].arrival\_time <= time && !is\_completed[i]) {

processes[i].waiting\_time = time - processes[i].arrival\_time;

time += processes[i].burst\_time;

processes[i].completion\_time = time;

is\_completed[i] = 1;

completion\_order[index++] = processes[i].id;

completed++;

process\_found = 1;

break; // Exit after finding the highest priority process

}

}

if (!process\_found) {

time++;

}

}

return 0;

}