

Stooge Sort (Придурковатая сортировка)

1. Общая постановка задачи

Необходимо создать новый компонент для калькулятора, который должен реализовывать новый интерфейс с методом сортировки Stooge sort (придурковатая сортировка). Интерфейс должен содержать методы с разными типами данных (int, long, float, double, long double).

2. Реализуемый алгоритм Stooge Sort

Stooge sort - это рекурсивный алгоритм сортировки, который имеет необычно высокую временную сложность $O(n^{2.7})$. Несмотря на свою неэффективность, алгоритм представляет интерес в учебных целях.

Алгоритм Stooge sort работает следующим образом:

Если первый элемент массива больше последнего, меняет их местами

Если в массиве более 2 элементов:

Рекурсивно сортирует первые $2/3$ массива

Рекурсивно сортирует последние $2/3$ массива

Снова рекурсивно сортирует первые $2/3$ массива

Алгоритм получает свое неофициальное название "придурковатая сортировка" из-за своей неэффективности и странной логики работы.

3. Асимптотика

Временная сложность:

Средняя: $O(n^{2.7})$

Лучшая: $O(n^{2.7})$

Худшая: $O(n^{2.7})$

Сложность по памяти: $O(n)$ - из-за глубины рекурсии

Алгоритм Stooge sort является одним из самых неэффективных алгоритмов сортировки, но его реализация достаточно проста и наглядна для понимания рекурсивных алгоритмов.

4. Реализация

При решении задачи интерфейс содержит 5 методов для различных типов данных:

```
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 *stoogeSortInt)(/* in */ IEcoLab1Ptr_t me, /* in */ int* array, /* in */ int size);  
    int16_t (ECOCALLMETHOD *stoogeSortLong)(/* in */ IEcoLab1Ptr_t me, /* in */ long* array, /* in */ int size);  
    int16_t (ECOCALLMETHOD *stoogeSortFloat)(/* in */ IEcoLab1Ptr_t me, /* in */ float* array, /* in */ int size);  
    int16_t (ECOCALLMETHOD *stoogeSortDouble)(/* in */ IEcoLab1Ptr_t me, /* in */ double* array, /* in */ int size);  
    int16_t (ECOCALLMETHOD *stoogeSortLongDouble)(/* in */ IEcoLab1Ptr_t me, /* in */ long double* array, /* in */ int size);  
}  
IEcoLab1VTbl, *IEcoLab1VTblPtr;
```

Каждый метод реализует рекурсивный алгоритм Stooge sort для соответствующего типа данных. Приведу пример основного и вспомогательного метода для целых чисел:

```
/*
 * Вспомогательные функции для Stooge sort
 */
static void stoogeSortHelperInt(int* arr, int l, int r) {
    if (arr[l] > arr[r]) {
        int temp = arr[l];
        arr[l] = arr[r];
        arr[r] = temp;
    }
    if (r - l + 1 > 2) {
        int t = (r - l + 1) / 3;
        stoogeSortHelperInt(arr, l, r - t);
        stoogeSortHelperInt(arr, l + t, r);
        stoogeSortHelperInt(arr, l, r - t);
    }
}
```

```
/*
 *
 * <сводка>
 * Функция stoogeSortInt
 * </сводка>
 *
 * <описание>
 * Функция сортирует массив целых чисел с помощью алгоритма Stooge sort
 * </описание>
 *
 */
static int16_t ECOCALLMETHOD CEcoLab1_stoogeSortInt(/* in */ IEcoLab1Ptr_t me, /* in */ int* array, /* in */ int size) {
    CEcoLab1* pCMe = (CEcoLab1*)me;
    /* Проверка указателей */
    if (me == 0 || array == 0 || size <= 0) {
        return ERR_ECO_POINTER;
    }
    stoogeSortHelperInt(array, 0, size - 1);

    return ERR_ECO_SUCCESSES;
}
```

5. Пример работы

```

Select C:\Windows\system32\cmd.exe
/// int32_t test ///
Test #1: -809 965 137 -719 866 694 693 -297 886 -627 402 175 650 -914 71 965 -328 -763 556 977 ...
Test took 5.000000 ms
Result #1: -991 -977 -937 -914 -864 -835 -821 -813 -809 -793 -782 -763 -750 -724 -719 -666 -633 -627 -622 -601 ...

Test #2: 924 -348 -210 855 750 840 -484 -540 377 -258 356 -654 -645 -899 779 -352 -490 604 -871 3 ...
Test took 347.000000 ms
Result #2: -1000 -994 -994 -993 -976 -973 -972 -972 -967 -961 -958 -946 -941 -940 -935 -932 -930 -926 -922 ...

Test #3: -590 -169 381 824 515 886 295 -943 -232 625 -992 346 910 -511 -919 -741 895 -230 -430 449 ...
Test took 1043.000000 ms
Result #3: -1000 -1000 -999 -997 -997 -996 -996 -995 -994 -992 -992 -988 -988 -985 -985 -984 -983 -977 -977 -977 ...

/// long test ///
Test #1: -955 -241 -280 60 -856 369 -735 -760 590 -194 739 -67 -473 613 416 965 -609 900 -887 80 ...
Test took 5.000000 ms
Result #1: -998 -980 -975 -965 -961 -955 -931 -905 -901 -887 -873 -856 -768 -760 -749 -735 -715 -707 -679 -650 ...

Test #2: 781 -774 -472 319 -586 911 129 328 876 314 830 167 264 -996 -286 337 670 79 987 631 ...
Test took 347.000000 ms
Result #2: -999 -999 -997 -996 -996 -985 -984 -982 -980 -980 -974 -966 -960 -958 -948 -943 -932 -929 -922 -919 ...

Test #3: 80 -614 48 -761 -44 -146 156 -784 -117 -273 -127 -828 132 -422 946 -174 299 -22 693 522 ...
Test took 1042.000000 ms
Result #2: -1000 -1000 -996 -995 -994 -991 -989 -987 -985 -982 -981 -980 -975 -973 -972 -971 -966 -963 -962 -960 ...

/// float test ///
Test #1: 116.55 -958.86 -490.04 753.29 940.18 30.79 600.70 -22.06 -251.44 240.27 -177.95 -497.18 -607.84 -701.10 744.07 244.00 -754.88 -281.78 930.66 -786.49 ...
Test took 5.000000 ms
Result #1: -986.21 -982.18 -979.92 -965.21 -958.86 -933.23 -925.60 -860.65 -845.15 -832.45 -826.29 -811.15 -801.63 -792.90 -791.80 -786.49 -772.39 -768.85 -762.26 -754.88 ...

Test #2: -613.76 44.83 845.27 -646.72 389.26 736.56 922.12 904.90 -326.88 -786.31 70.35 -978.51 736.93 -511.64 -272.26 -31.22 46.72 -626.33 -181.37 881.04 ...
Test took 365.000000 ms
Result #2: -999.69 -991.64 -987.18 -982.06 -978.51 -971.13 -969.36 -965.64 -962.22 -959.04 -957.82 -951.66 -951.05 -949.40 -947.69 -946.59 -934.87 -933.77 -932.98 -929.56 ...

Test #3: -397.69 -683.28 -117.22 -159.28 -353.56 248.27 212.81 527.88 -599.23 994.87 -426.86 759.76 426.86 771.23 -378.40 657.46 77.06 -627.12 413.86 726.92 ...
Test took 1098.000000 ms
Result #2: -996.58 -995.79 -992.74 -988.34 -987.91 -986.88 -986.69 -985.53 -982.54 -979.92 -979.25 -970.09 -969.73 -963.87 -956.54 -956.48 -954.83 -949.58 -948.91 -948.12 ...

Test took 4.000000 ms
Result #1: -979.67 -968.87 -965.58 -959.41 -953.37 -939.70 -907.59 -900.75 -898.74 -864.50 -838.37 -804.01 -792.72 -763.18 -742.06 -696.04 -677.54 -676.08 -645.56 -640.74 ...

Test #2: 746.76 -409.53 389.69 988.16 243.02 213.42 -560.53 320.96 -917.66 326.03 -223.73 -412.21 -451.89 836.97 397.69 -375.41 644.95 -805.54 572.92 781.00 ...
Test took 368.000000 ms
Result #2: -998.84 -995.91 -994.81 -978.51 -976.38 -975.65 -970.70 -962.95 -961.61 -955.57 -942.81 -941.28 -935.12 -932.25 -928.34 -923.28 -917.66 -913.14 -908.32 -906.49 ...

Test #3: -426.37 -734.86 512.99 336.47 -437.18 600.82 -87.19 -32.56 -974.24 -398.54 -628.65 323.71 -578.30 -961.36 125.64 38.42 817.19 -184.24 -841.55 86.52 ...
Test took 1105.000000 ms
Result #2: -998.05 -996.83 -996.77 -995.12 -990.60 -988.77 -985.47 -983.64 -982.91 -982.60 -974.24 -968.63 -967.22 -963.74 -961.36 -961.00 -957.64 -955.75 -953.92 -953.55 ...

/// long double test ///
Test #1: 923.826 -198.950 -190.039 -260.781 -550.401 725.272 -26.032 -886.959 -296.854 921.690 267.739 321.268 -404.157 -663.442 810.480 -868.282 445.723 -850.154 -339.213 -427.046 ...
Test took 5.000000 ms
Result #1: -959.838 -943.175 -929.563 -886.959 -876.217 -868.282 -850.154 -817.072 -815.912 -779.595 -720.328 -711.173 -709.708 -709.586 -663.442 -617.359 -588.549 -582.018 -557.237 -550.401 ...

Test #2: -158.361 260.964 -424.604 652.089 -850.215 -890.744 -34.333 913.999 -648.488 -313.395 252.968 -704.947 577.013 610.828 557.299 514.512 431.623 777.764 55.391 449.324 ...
Test took 366.000000 ms
Result #2: -999.207 -997.497 -996.399 -996.277 -993.225 -989.074 -983.886 -982.482 -981.384 -979.980 -978.393 -977.599 -974.853 -971.435 -968.444 -964.171 -959.593 -959.349 -957.213 -955.748 ...

Test #3: 659.536 -552.049 -294.412 -652.028 -677.480 -748.589 -765.862 996.277 523.606 667.531 -443.342 -570.666 -317.728 -162.816 52.278 104.892 641.591 -416.242 193.030 -220.740 ...
Test took 1101.000000 ms
Result #2: -999.756 -997.009 -990.967 -986.511 -986.511 -986.267 -986.267 -975.951 -969.909 -964.965 -962.462 -959.471 -959.410 -958.922 -956.481 -953.551 -953.185 -952.696 -950.438 -949.278 ...

Press any key to continue . . . █
```

Программа работает с типами данных int, long, float, double, long double. Алгоритм корректно сортирует как положительные, так и отрицательные числа.

6. Сравнение с qsort из библиотеки stdlib.h

Результаты работы stoogeSortInt, stoogeSortLong, stoogeSortFloat, stoogeSortDouble и stoogeSortLongDouble в миллисекундах на 100, 500 и 1000 элементов с пятью типами данных:

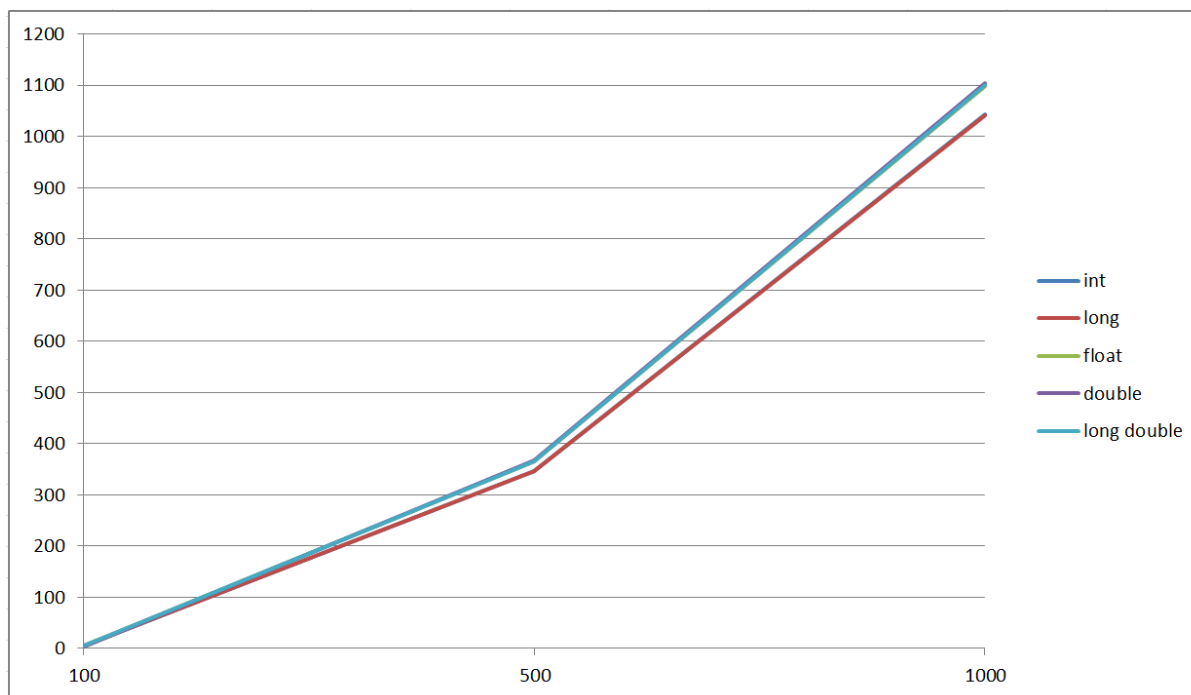
My StoogeSort					
	int	long	float	double	long double
100	5	5	5	4	5
500	347	347	365	368	366
1000	1043	1042	1098	1105	1101

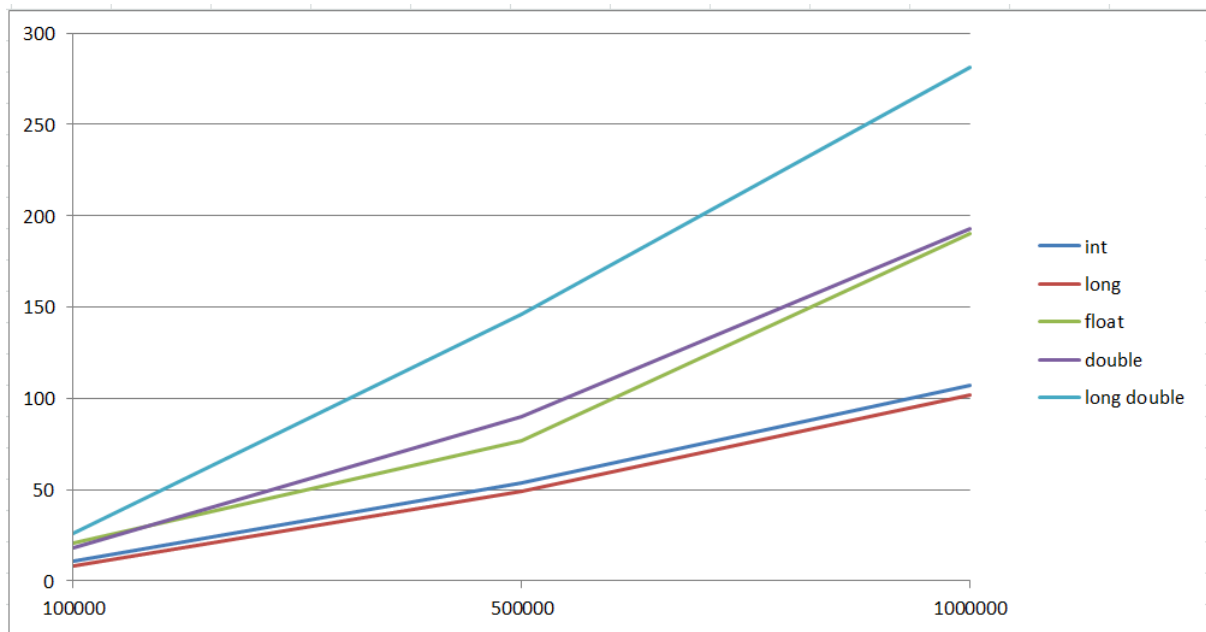
Результаты работы стандартной функции qsort из библиотеки stdlib.h в миллисекундах на 100000, 500000 и 1000000 элементов с пятью типами данных:

stdlib.h qsort					
	int	long	float	double	long double
100000	11	8	21	18	26
500000	54	49	77	90	146
1000000	107	102	190	193	281

Можно заметить, что алгоритм Stooge sort значительно уступает по производительности стандартной функции qsort, особенно при увеличении размера массива. Это ожидаемо, учитывая асимптотическую сложность $O(n^{2.7})$ против $O(n \log n)$ у qsort.

Также сравним диаграммы:





Разница в производительности становится критической при обработке массивов размером более 1000 элементов. Для массива из 1000 элементов Stooge sort работает примерно в 10 раз медленнее, чем qsort для массива из 1 000 000 элементов.

7. Выводы

Алгоритм Stooge sort представляет собой интересный учебный пример рекурсивного алгоритма, но его практическое применение ограничено из-за высокой временной сложности. Тем не менее, реализация этого алгоритма в рамках компонентной архитектуры Есо позволяет понять принципы работы с различными типами данных и демонстрирует важность выбора эффективного алгоритма для решения конкретной задачи.