

Реализация стандартных сортировок в языках C++, Python

Артём Бислюк,
Дмитрий Горбач,
БГУ, ФПМИ, 2020 год

Реализация `std::sort(...)` в C++ (MinGW)

// Основная функция

```
void sort(Iterator first, Iterator last, Comparator comp) {  
    if (first != last) {  
        // вызывает функцию, выбирающую тип сортировки (qsort или кучей)  
        introsort_loop(first, last, log(last - first) * 2, comp);  
        // "досортировывает" массив вставками  
        final_insertion_sort(first, last, comp);  
    }  
}
```

std::__introsort_loop(...) в C++ (MinGW)

```
void introsort_loop(Iterator first, Iterator last, Size depth_limit,
Comparator comp) {
    // выполняется, пока неотсортированных элементов больше 16
    while (last - first > 16) {
        // проверяет, не превышен ли лимит операций для qsort
        if (depth_limit == 0) {
            // если лимит операций превышен, запускается сортировка кучей
            partial_sort(first, last, last, comp);
            return;
        }
        // очередная итерация qsort
        --depth_limit;
        Iterator cut = unguarded_partition_pivot(first, last, comp);
        introsort_loop(cut, last, depth_limit, comp);
        last = cut;
    }
}
```

std::__insertion_sort(...) в C++ (MinGW)

```
void insertion_sort(Iterator first, Iterator last, Compare comp) {  
    if (first == last)  
        return;  
    // выполняем сортировку вставками  
    for (auto i = first + 1; i != last; ++i) {  
        if (!comp(i, first)) {  
            // вызов функции для вставки элемента  
            unguarded_linear_insert(i, comp);  
        }  
    }  
}
```

Оценка сложности используемых сортировок

вид сортировки	лучший случай	средний случай	худший случай
qsort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
кучей	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
вставками	$O(n)$ (0 обменов)	$O(n^2)$	$O(n^2)$

Интересный факт. Хотя сортировка кучей кажется лучше, чем остальные, однако этот у неё есть ряд недостатков:

1. Не работает на списках, так как требует доступ за $O(1)$ к элементу контейнера.
2. Не распараллеливается.

Реальная реализация на C++ (MinGW)

(для справки)

C++ — реальная реализация std::__sort(...)

```
template<typename _RandomAccessIterator, typename _Compare>
inline void __sort(_RandomAccessIterator __first, _RandomAccessIterator __last, _Compare __comp) {
    if (__first != __last) {
        std::__introsort_loop(__first, __last, std::__lg(__last - __first) * 2, __comp);
        std::__final_insertion_sort(__first, __last, __comp);
    }
}
```

C++ — реальная реализация std::__introsort_loop(...) и вспомогательных функций

```
template<typename _RandomAccessIterator, typename _Size, typename _Compare>
void __introsort_loop(_RandomAccessIterator __first, _RandomAccessIterator __last, _Size __depth_limit, _Compare __comp) {
    while (__last - __first > int(_S_threshold)) {
        if (__depth_limit == 0) {
            std::__partial_sort(__first, __last, __last, __comp);
            return;
        }
        --__depth_limit;
        _RandomAccessIterator __cut = std::__unguarded_partition_pivot(__first, __last, __comp);
        std::__introsort_loop(__cut, __last, __depth_limit, __comp);
        __last = __cut;
    }
}

//heap sort
template<typename _RandomAccessIterator, typename _Compare>
inline void __partial_sort(_RandomAccessIterator __first, _RandomAccessIterator __middle, _RandomAccessIterator __last, _Compare __comp) {
    std::__heap_select(__first, __middle, __last, __comp);
    std::__sort_heap(__first, __middle, __comp);
}

//just moves bounds (divides by 2)
template<typename _RandomAccessIterator, typename _Compare>
inline _RandomAccessIterator __unguarded_partition_pivot(_RandomAccessIterator __first,
    _RandomAccessIterator __last, _Compare __comp) {
    _RandomAccessIterator __mid = __first + (__last - __first) / 2;
    std::__move_median_to_first(__first, __first + 1, __mid, __last - 1,
        __comp);
    return std::__unguarded_partition(__first + 1, __last, __first, __comp);
}
```


C++ — реальная реализация std::__final_insertion_sort(...) и вспомогательных функций

```
template<typename _RandomAccessIterator, typename _Compare>
void __final_insertion_sort(_RandomAccessIterator __first, _RandomAccessIterator __last, _Compare __comp) {
    if (__last - __first > int(_S_threshold)) {
        std::__insertion_sort(__first, __first + int(_S_threshold), __comp);
        std::__unguarded_insertion_sort(__first + int(_S_threshold), __last, __comp);
    } else
        std::__insertion_sort(__first, __last, __comp);
}

template<typename _RandomAccessIterator, typename _Compare>
void __insertion_sort(_RandomAccessIterator __first, _RandomAccessIterator __last, _Compare __comp) {
    if (__first == __last) return;
    for (_RandomAccessIterator __i = __first + 1; __i != __last; ++__i) {
        if (__comp(__i, __first)) {
            typename iterator_traits<_RandomAccessIterator>::value_type __val = _GLIBCXX_MOVE(*__i);
            _GLIBCXX_MOVE_BACKWARD3(__first, __i, __i + 1);
            *__first = _GLIBCXX_MOVE(__val);
        } else
            std::__unguarded_linear_insert(__i, __gnu_cxx::__ops::__val_comp_iter(__comp));
    }
}

template<typename _RandomAccessIterator, typename _Compare>
inline void __unguarded_insertion_sort(_RandomAccessIterator __first, _RandomAccessIterator __last, _Compare __comp) {
    for (_RandomAccessIterator __i = __first; __i != __last; ++__i)
        std::__unguarded_linear_insert(__i, __gnu_cxx::__ops::__val_comp_iter(__comp));
}

template<typename _RandomAccessIterator, typename _Compare>
void __unguarded_linear_insert(_RandomAccessIterator __last, _Compare __comp) {
    typename iterator_traits<_RandomAccessIterator>::value_type __val = _GLIBCXX_MOVE(*__last);
    _RandomAccessIterator __next = __last;
    --__next;
    while (__comp(__val, __next)) {
        *__last = _GLIBCXX_MOVE(*__next);
        __last = __next;
        --__next;
    }
    *__last = _GLIBCXX_MOVE(__val);
}
```


Алгоритм TimSort — основная идея



- Этап 1.** По специальному алгоритму входной массив разделяется на подмассивы.
- Этап 2.** Каждый подмассив сортируется сортировкой вставками.
- Этап 3.** Отсортированные подмассивы собираются в единый массив с помощью модифицированной сортировки слиянием.

лучший случай	средний случай	худший случай
$O(n)$	$O(n \log n)$	$O(n \log n)$

Вычисление минимального размера подмассива



```
def calc_minrun(n: int):  
    r = 0  
    while n >= 64:  
        r = r | n & 1  
        n = n >> 1  
    return n + r
```

Этап 1. Разбиение на подмассивы, их сортировка



```
def divide_and_sort(arr : list, minrun : int):  
    i = 0  
    while i < len(arr):  
        # run - максимальный упорядоченный подмассив, начиная с i  
        run_end = find_run_end(i)  
        # если размер run меньше minrun, добавляем minrun - run_len  
        # элементов  
        if run_len < minrun:  
            run_len += minrun - run_len  
        # применяем к данному подмассиву сортировку вставками  
        insertion_sort(arr, i, run_len)  
        # возвращаем пару (индекс начала подмассива, длина подмассива)  
        yield (i, run_len)  
        i += run_len
```

Этап 2. Слияние

subs_info получили на этапе 1

```
def merge_sort(arr : list, subs_info : list):
```

```
    # стэк пар (индекс начала подмассива, длина подмассива)
```

```
    s = stack()
```

```
    s.append(subs_info[0])
```

```
    s.append(subs_info[1])
```

```
    i = 2
```

```
    while len(s) != 0:
```

```
        if i < len(subs_info):
```

```
            s.append(subs_info[i])
```

```
            i += 1
```

```
    X = первый подмассив в стэке
```

```
    Y = второй подмассив в стэке
```

```
    Z = третий подмассив в стэке
```

```
    if not (len(Z) > len(Y) + len(X) and len(Y) > len(X)):
```

```
        merge(Y, min(X, Z))
```

Модификация слияния — Galloping mode



1. Начинается обычная **сортировка слиянием**.
2. При переносе элемента в результирующий массив **запоминается** подмассив, откуда был взят элемент
3. **Если** большое число элементов было уже взято из одного подмассива,
 - назовем данный подмассив **поставщиком**
 - переходим в **Galloping mode** — перемещаемся по этому массиву **бинарным поиском**.
4. **Если** данные из поставщика больше не подходят или достигнут конец
 - копируем все данные до последнего подходящего элемента
 - выходим из “режима галопа”

Реальная реализация сортировки в CPython

<https://github.com/python/cpython/blob/master/Objects/listobject.c>

Функция `list_sort_impl()`

