Сортировка слиянием

Сортировка слиянием (англ. *merge sort*) — алгоритм сортировки, который упорядочивает списки (или другие структуры данных, доступ к элементам которых можно получать только последовательно, например — потоки) в определённом порядке. Эта сортировка — хороший пример использования принципа «разделяй и властвуй». Сначала задача разбивается на несколько подзадач меньшего размера. Затем эти задачи решаются с помощью рекурсивного вызова или непосредственно, если их размер достаточно мал. Наконец, их решения комбинируются, и получается решение исходной задачи.

Сортировка слиянием

Пример сортировки слиянием. Сначала делим список на кусочки (по 1 элементу), затем сравниваем каждый элемент с соседним, сортируем и объединяем. В итоге, все элементы отсортированы и объединены вместе.

Автор	Джон фон Нейман
Предназначение	Алгоритм сортировки
Структура данных	Массив
Худшее время	O($n \log_2 n$)
Лучшее время	O($n \log_2 n$)
Среднее время	O($n \log_2 n$)
Затраты памяти	O(n) вспомогательных

Содержание

Подробный алгоритм сортировки

Для решения задачи сортировки эти три этапа выглядят так:

1. Сортируемый массив разбивается на две части примерно одинакового размера;
2. Каждая из получившихся частей сортируется отдельно, например — тем же самым алгоритмом;
3. Два упорядоченных массива половинного размера соединяются в один.

Действие алгоритма на примере сортировки случайных точек.

1.1. — 2.1. Рекурсивное разбиение задачи на меньшие происходит до тех пор, пока размер массива не достигнет единицы (любой массив длины 1 можно считать упорядоченным).

3.1. Соединение двух упорядоченных массивов в один.
Основную идею слияния двух отсортированных массивов можно объяснить на следующем примере. Пусть мы имеем два уже отсортированных по возрастанию подмассива. Тогда:
3.2. Слияние двух подмассивов в третий результирующий массив.
На каждом шаге мы берём меньший из двух первых элементов подмассивов и записываем его в результирующий массив. Счётчики номеров элементов результирующего массива и подмассива, из которого был взят элемент, увеличиваем на 1.
3.4. «Прицепление» остатка.
Когда один из подмассивов закончился, мы добавляем все оставшиеся элементы второго подмассива в результирующий массив.

Пример сортировки на языке C

```c
 /**
  * Сортирует массив, используя рекурсивную сортировку слиянием
  * up - указатель на массив, который нужно сортировать
  * down - указатель на массив с, как минимум, таким же размером как у 'up', используется как буфер
  * left - левая граница массива, передайте 0, чтобы сортировать массив с начала
  * right - правая граница массива, передайте длину массива - 1, чтобы сортировать массив до последнего элемента
  * возвращает: указатель на отсортированный массив. Из-за особенностей работы данной реализации
  * отсортированная версия массива может оказаться либо в 'up', либо в 'down'
  **/
int* merge_sort(int *up, int *down, unsigned int left, unsigned int right)
{
    if (left == right)
    {
        down[left] = up[left];
        return down;
    }

    unsigned int middle = (left + right) / 2;

    // разделяй и сортируй
    int *l_buff = merge_sort(up, down, left, middle);
    int *r_buff = merge_sort(up, down, middle + 1, right);

    // слияние двух отсортированных половин
    int *target = l_buff == up ? down : up;

    unsigned int l_cur = left, r_cur = middle + 1;
    for (unsigned int i = left; i <= right; i++)
    {
        if (l_cur <= middle && r_cur <= right)
        {
            if (l_buff[l_cur] < r_buff[r_cur])
            {
                target[i] = l_buff[l_cur];
                l_cur++;
            }
            else
            {
                target[i] = r_buff[r_cur];
                r_cur++;
            }
        }
        else if (l_cur <= middle)
        {
            target[i] = l_buff[l_cur];
            l_cur++;
        }
        else
        {
            target[i] = r_buff[r_cur];
            r_cur++;
        }
    }
    return target;
}
```

Реализация на языке C++11:

```cpp
#include <algorithm>
#include <cstddef>
#include <iterator>
#include <memory>

template<typename T>
void merge_sort(T array[], std::size_t size) noexcept
{
    if (size > 1)
    {
        std::size_t const left_size = size / 2;
        std::size_t const right_size = size - left_size;

        merge_sort(&array[0], left_size);
        merge_sort(&array[left_size], right_size);

        std::size_t lidx = 0, ridx = left_size, idx = 0;
        std::unique_ptr<T[]> tmp_array(new T[size]);

        while (lidx < left_size || ridx < size)
        {
            if (array[lidx] < array[ridx])
            {
                tmp_array[idx++] = std::move(array[lidx]);
                lidx++;
```

```
            }
            else
            {
                tmp_array[idx++] = std::move(array[ridx]);
                ridx++;
            }

            if (lidx == left_size)
            {
                std::copy(std::make_move_iterator(&array[ridx]),
                          std::make_move_iterator(&array[size]),
                          &tmp_array[idx]);
                break;
            }
            if (ridx == size)
            {
                std::copy(std::make_move_iterator(&array[lidx]),
                          std::make_move_iterator(&array[left_size]),
                          &tmp_array[idx]);
                break;
            }
        }

        std::copy(std::make_move_iterator(tmp_array),
                  std::make_move_iterator(&tmp_array[size]),
                  array);
    }
}
```

Реализация на языке C++14 с распараллеливанием от OpenMP

```
 1 #include <algorithm>
 2 #include <iterator>
 3 #include <omp.h>
 4 #include <memory>
 5
 6 template <typename Iterator>
 7 void mergesort(Iterator from, Iterator to)
 8 {
 9 #pragma omp parallel
10     {
11 #pragma omp single nowait
12         static_assert(!std::is_same<typename std::iterator_traits<Iterator>::value_type, void>::value);
13
14         auto n = std::distance(from, to);
15
16         if (1 < n)
17         {
18 #pragma omp task firstprivate (from, to, n)
19             {
20                 Iterator l_from = from;
21                 Iterator l_to = l_from;
22                 std::advance(l_to, n/2);
23                 mergesort(l_from, l_to);
24             }
25 #pragma omp task firstprivate (from, to, n)
26             {
27                 Iterator r_from = from;
28                 std::advance(r_from, n/2);
29                 Iterator r_to = r_from;
30                 std::advance(r_to, n-(n/2));
31                 mergesort(r_from, r_to);
32             }
33 #pragma omp taskwait
34
35             auto tmp_array = std::make_unique<typename Iterator::value_type[]>(n);
36             Iterator l_iter = from;
37             Iterator l_end = l_iter;
38             std::advance(l_end, n/2);
39             Iterator r_iter = l_end;
40             Iterator& r_end = to;
41
42             auto tmp_iter = tmp_array.get();
43
44             while (l_iter != l_end || r_iter != r_end)
45             {
46                 if (*l_iter < *r_iter)
47                 {
48                     *tmp_iter = std::move(*l_iter);
49                     ++l_iter;
50                     ++tmp_iter;
51                 }
52                 else
53                 {
54                     *tmp_iter = std::move(*r_iter);
55                     ++r_iter;
```

```
                    ++tmp_iter;
                }

                if (l_iter == l_end)
                {
                    std::copy(
                                std::make_move_iterator(r_iter),
                                std::make_move_iterator(r_end),
                                tmp_iter
                    );

                    break;
                }

                if (r_iter == r_end)
                {
                    std::copy(
                                std::make_move_iterator(l_iter),
                                std::make_move_iterator(l_end),
                                tmp_iter
                    );

                    break;
                }
            }

            std::copy(
                        std::make_move_iterator(tmp_array.get()),
                        std::make_move_iterator(&tmp_array[n]),
                        from
            );
        }
    }
}
```

Итеративная реализация на языке C++:

```
template<typename T>
void MergeSort(T a[], size_t l)
{
    size_t BlockSizeIterator;
    size_t BlockIterator;
    size_t LeftBlockIterator;
    size_t RightBlockIterator;
    size_t MergeIterator;

    size_t LeftBorder;
    size_t MidBorder;
    size_t RightBorder;
    for (BlockSizeIterator = 1; BlockSizeIterator < l; BlockSizeIterator *= 2)
    {
        for (BlockIterator = 0; BlockIterator < l - BlockSizeIterator; BlockIterator += 2 * BlockSizeIterator)
        {
            //Производим слияние с сортировкой пары блоков начинающуюся с элемента BlockIterator
            //левый размером BlockSizeIterator, правый размером BlockSizeIterator или меньше
            LeftBlockIterator = 0;
            RightBlockIterator = 0;
            LeftBorder = BlockIterator;
            MidBorder = BlockIterator + BlockSizeIterator;
            RightBorder = BlockIterator + 2 * BlockSizeIterator;
            RightBorder = (RightBorder < l) ? RightBorder : l;
            int* SortedBlock = new int[RightBorder - LeftBorder];

            //Пока в обоих массивах есть элементы выбираем меньший из них и заносим в отсортированный блок
            while (LeftBorder + LeftBlockIterator < MidBorder && MidBorder + RightBlockIterator < RightBorder)
            {
                if (a[LeftBorder + LeftBlockIterator] < a[MidBorder + RightBlockIterator])
                {
                    SortedBlock[LeftBlockIterator + RightBlockIterator] = a[LeftBorder + LeftBlockIterator];
                    LeftBlockIterator += 1;
                }
                else
                {
                    SortedBlock[LeftBlockIterator + RightBlockIterator] = a[MidBorder + RightBlockIterator];
                    RightBlockIterator += 1;
                }
            }
            //После этого заносим оставшиеся элементы из левого или правого блока
            while (LeftBorder + LeftBlockIterator < MidBorder)
            {
                SortedBlock[LeftBlockIterator + RightBlockIterator] = a[LeftBorder + LeftBlockIterator];
                LeftBlockIterator += 1;
            }
            while (MidBorder + RightBlockIterator < RightBorder)
            {
                SortedBlock[LeftBlockIterator + RightBlockIterator] = a[MidBorder + RightBlockIterator];
```

```
                RightBlockIterator += 1;
            }

            for (MergeIterator = 0; MergeIterator < LeftBlockIterator + RightBlockIterator; MergeIterator++)
            {
                a[LeftBorder + MergeIterator] = SortedBlock[MergeIterator];
            }
            delete SortedBlock;
        }
    }
}
```

Псевдокод алгоритма слияния без «прицепления» остатка на C++-подобном языке:

```
L = *In1;
R = *In2;
if( L == R ) {
 *Out++ = L;
 In1++;
 *Out++ = R;
 In2++;
} else if(L < R) {
 *Out++ = L;
 In1++;
} else {
 *Out++ = R;
 In2++;
}
```

Алгоритм был изобретён Джоном фон Нейманом в 1945 году[1].

В приведённом алгоритме на C++-подобном языке используется проверка на равенство двух сравниваемых элементов подмассивов с отдельным блоком обработки в случае равенства. Отдельная проверка на равенство удваивает число сравнений, что усложняет код программы. Вместо отдельной проверки на равенство и отдельного блока обработки в случае равенства можно использовать две проверки **if(L <= R)** и **if(L >= R)**, что почти вдвое уменьшает код программы.

Псевдокод улучшенного алгоритма слияния без «прицепления» остатка на C++-подобном языке:

```
L = *In1;
R = *In2;

if( L <= R ) {
 *Out++ = L;
 In1++;
}
if( L >= R ) {
 *Out++ = R;
 In2++;
}
```

Число проверок можно сократить вдвое убрав проверку **if(L >= R)**. При этом, в случае равенства L и R, L запишется в Out в первой итерации, а R - во второй. Этот вариант будет работать эффективно, если в исходном массиве повторяющиеся элементы не будут преобладать над остальными элементами.

Псевдокод сокращенного алгоритма слияния без «прицепления» остатка на C++-подобном языке:

```
L = *In1;
R = *In2;

if( L <= R ) {
 *Out++ = L;
 In1++;
} else {
 *Out++ = R;
 In2++;
}
```

Две операции пересылки в переменные **L** и **R** упрощают некоторые записи в программе, что может оказаться полезным в учебных целях, но в действительных программах их можно удалить, что сократит программный код. Также можно использовать тернарный оператор, что еще больше сократит программный код.
Псевдокод ещё более улучшенного алгоритма слияния без «прицепления» остатка на C++-подобном языке:

```
*Out++ =  *In1 <= *In2  ?  In1++ : In2++;
```

Время работы алгоритма порядка O(n * log n) при отсутствии деградации на неудачных случаях, которая является больным местом быстрой сортировки (тоже алгоритм порядка O(n * log n), но только для среднего случая). Расход памяти выше, чем для быстрой сортировки, при намного более благоприятном паттерне выделения памяти — возможно выделение одного региона памяти с самого начала и отсутствие выделения при дальнейшем исполнении.

Популярная реализация требует однократно выделяемого временного буфера памяти, равного сортируемому массиву, и не имеет рекурсий. Шаги реализации:

1. InputArray = сортируемый массив, OutputArray = временный буфер
2. над каждым отрезком входного массива InputArray[N * MIN_CHUNK_SIZE..(N + 1) * MIN_CHUNK_SIZE] выполняется какой-то вспомогательный алгоритм сортировки, например, сортировка Шелла или быстрая сортировка.
3. устанавливается ChunkSize = MIN_CHUNK_SIZE
4. сливаются два отрезка InputArray[N * ChunkSize..(N + 1) * ChunkSize] и InputArray[(N + 1) * ChunkSize..(N + 2) * ChunkSize] попеременным шаганием слева и справа (см. выше), результат помещается в OutputArray[N * ChunkSize..(N + 2) * ChunkSize], и так для всех N, пока не будет достигнут конец массива.
5. ChunkSize удваивается
6. если ChunkSize стал >= размера массива, то конец, результат в OutputArray, который (ввиду перестановок, описанных ниже) есть либо сортируемый массив, либо временный буфер, во втором случае он целиком копируется в сортируемый массив.
7. иначе меняются местами InputArray и OutputArray перестановкой указателей, и всё повторяется с пункта 4.

Такая реализация также поддерживает размещение сортируемого массива и временного буфера в дисковых файлах, то есть пригодна для сортировки огромных объёмов данных. Реализация ORDER BY в СУБД MySQL при отсутствии подходящего индекса устроена именно так (источник: filesort.cc в исходном коде MySQL).

Пример реализации алгоритма простого двухпутевого слияния на псевдокоде:

```
function mergesort(m)
    var list left, right, result
    if length(m) ≤ 1
        return m
    else
        middle = length(m) / 2
        for each x in m up to middle
            add x to left
        for each x in m after middle
            add x to right
        left = mergesort(left)
        right = mergesort(right)
        result = merge(left, right)
        return result
    end if
```

Есть несколько вариантов функции merge(), наиболее простой вариант может выглядеть так:

```
function merge(left,right)
    var list result
    while length(left) > 0 and length(right) > 0
        if first(left) ≤ first(right)
            append first(left) to result
            left = rest(left)
        else
            append first(right) to result
            right = rest(right)
        end if
    while length(left) > 0
        append first(left) to result
        left = rest(left)
    while length(right) > 0
        append first(right) to result
        right = rest(right)
    return result
```

Достоинства и недостатки

Достоинства:

- Работает даже на структурах данных последовательного доступа.
- Хорошо сочетается с подкачкой и кэшированием памяти.
- Неплохо работает в параллельном варианте: легко разбить задачи между процессорами поровну, но трудно сделать так, чтобы другие процессоры взяли на себя работу, в случае если один процессор задержится.
- Не имеет «трудных» входных данных.
- Устойчивая - сохраняет порядок равных элементов (принадлежащих одному классу эквивалентности по сравнению).

Недостатки:

- На «почти отсортированных» массивах работает столь же долго, как на хаотичных. Существует вариант сортировки слиянием, который работает быстрее на частично отсортированных данных, но он требует дополнительной памяти, в дополнении ко временному буферу, который используется непосредственно для сортировки.
- Требует дополнительной памяти по размеру исходного массива.

Примечания

1. *Knuth, D.E.* The Art of Computer Programming. Volume 3: Sorting and Searching (англ.). — 2nd. — Addison-Wesley, 1998. — P. 159. — ISBN 0-201-89685-0.

Литература

- *Левитин А. В.* Глава 4. Метод декомпозиции: Сортировка слиянием // Алгоритмы. Введение в разработку и анализ — М.: Вильямс, 2006. — С. 169–172. — 576 с. — ISBN 978-5-8459-0987-9
- *Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, К.* Алгоритмы: построение и анализ = Introduction to Algorithms / Под ред. И. В. Красикова. — 2-е изд. — М.: Вильямс, 2005. — 1296 с. — ISBN 5-8459-0857-4.

Ссылки

- Многофазное слияние на algolist.manual.ru (http://algolist.manual.ru/sort/faq/q13.php)
- Сортировка слиянием (https://web.archive.org/web/20090523191746/http://iproc.ru/parallel-programming/lection-6/) — восходящая сортировка, естественная сортировка, измерение быстродействия.
- Описание метода и листинг программ сортировки слиянием (http://kvodo.ru/mergesort.html).
- Динамическая визуализация 7 алгоритмов сортировки с открытым исходным кодом (https://airtucha.github.io/SortVis/)
- Пример реализации на Java (https://urvanov.ru/2017/08/19/%d0%b0%d0%bb%d0%b3%d0%be%d1%80%d0%b8%d1%82%d0%bc-%d1%81%d0%be%d1%80%d1%82%d0%b8%d1%80%d0%be%d0%b2%d0%ba%d0%b8-%d1%81%d0%bb%d0%b8%d1%8f%d0%bd%d0%b8%d0%b5%d0%bc-%d0%bd%d0%b0-java/)
