

# TimSort

Владимир Огородников

1 ноября 2016 г.

## 1 Постановка задачи

Требуется отсортировать массив произвольных данных длины  $n$  за время  $O(n \log n)$ .

## 2 Алгоритм

### 2.1 Описание алгоритма

Разобьём массив на последовательные блоки так, чтобы в каждом из них элементы были расположены в строго убывающем либо нестрого возрастающем порядке, и при этом длина каждого блока была не меньше, чем  $f(n) \leq C$  (см. реализацию в следующем разделе). Для этого воспользуемся жадным алгоритмом: предположим, что первые  $k$  элементов массива уже разделены на блоки. Объединим следующие  $\min(f(n), n - k)$  элементов в блок. Если они уже упорядочены, продолжим добавлять элементы так, чтобы заданная монотонность сохранялась, после чего развернём блок так, если элементы в нём убывают. В противном случае воспользуемся любой сортировкой, например, `insertionSort()`.

Заведём список блоков и добавим в него *сверхжирный*, *жирный* и все нормальные блоки в естественном порядке. Сверхжирным блоком будем называть фиктивный блок длины  $2n + 2$ , жирным —  $n + 1$ . Сольём некоторые блоки так, чтобы для любых двух последовательных блоков длиной  $Y$  и  $Z$  выполнялось  $Y > Z$  (свойство 1), а для любых трёх  $X$ ,  $Y$  и  $Z$  выполнялось  $X > Y + Z$  (свойство 2). Заметим, что для первой тройки и для первых двух двоек эти свойства выполнены всегда вне зависимости от размера первого блока. Будем рассматривать блоки слева направо, начиная с четвёртого (с индексом 3 в 0-индексации). Если для него и предыдущего нарушено свойство 1, сольём его с предыдущим с помощью `inplaceMerge()`, после чего рекурсивно запустимся от соответствующего элемента списка. Если для него и двух предыдущих нарушено свойство 2, сольём два предыдущих блока, после чего рекурсивно запустимся от результата слияния, после чего рекурсивно запустимся от исходно рассматриваемого блока. Если оба свойства выполнены, перейдём к следующему блоку.

Теперь начнём сливать получившиеся блоки с конца списка. Когда в списке останется 1 нормальный блок (и 2 фиктивных), массив окажется отсортированным. Конец алгоритма.

### 2.2 Код на C++

Данная версия кода может быть устаревшей. Последнюю версию можно найти на <https://github.com/ogorodnikoff2012/TimSort>.

```
1  #ifndef TIMSORT_TIMSORT_H
2  #define TIMSORT_TIMSORT_H
3
4  #include "list.h"
5  #include "utils.h"
6  #include "itertools.h"
7  #include "merge.h"
8  #include "inplaceMerge.h"
9  #include "slowsort.h"
10 #include "params.h"
11 #include "debug.h"
12
13 #include <iterator>
14
15 namespace timsort {
16     template <class RAIterator>
```

```

17     struct TRun {
18         RAIterator begin, end;
19         std::size_t dummyLength;
20         TRun(RAIterator begin, RAIterator end, std::size_t dummyLength = 0) : begin(begin),
21             end(end), dummyLength(dummyLength) { }
22         TRun(const TRun<RAIterator> &run) : begin(run.begin), end(run.end), dummyLength(run.dummyLength)
23         std::size_t length() const {
24             return dummyLength ? dummyLength : std::distance(begin, end);
25         }
26         bool dummy() const {
27             return dummyLength; // != 0
28         }
29     };
30
31     template <class RAIterator>
32     std::ostream &operator <<(std::ostream &o, const TRun<RAIterator> &run) {
33         o << "Length: " << run.length() << ' ';
34         if (run.dummy()) {
35             o << "<dummy>";
36         } else {
37             // __ostr_iter<RAIterator>(o, run.begin) << run.end;
38         }
39         return o;
40     }
41
42     template <class RAIterator, class Comparator>
43     void appendRun(List<TRun<RAIterator>> &runList,
44         typename List<TRun<RAIterator>>::iterator newRun, Comparator cmp,
45         const ITimSortParams &params) {
46         #ifdef DEBUG
47             IT_PRINT(Runs, runList.begin(), ++newRun);
48             --newRun;
49         #endif
50         auto itZ = newRun, itY = newRun, itX = newRun;
51         std::advance(itY, -1);
52         std::advance(itX, -2);
53
54         EWhatMerge whatMerge;
55
56         if (itY->dummy()) {
57             whatMerge = WM_NoMerge;
58         } else if (itX->dummy()) {
59             if (params.needMerge(itY->length(), itZ->length())) {
60                 whatMerge = WM_MergeYZ;
61             } else {
62                 whatMerge = WM_NoMerge;
63             }
64         } else {
65             whatMerge = params.whatMerge(itX->length(), itY->length(), itZ->length());
66         }
67
68         switch (whatMerge) {
69             case WM_NoMerge:
70                 break;
71             case WM_MergeYZ:
72                 inplaceMerge(itY->begin, itY->end, itZ->end, cmp, params);
73                 itZ->begin = itY->begin;
74                 runList.erase(itY);
75                 appendRun(runList, itZ, cmp, params);
76                 break;
77             case WM_MergeXY:

```

```

78         inplaceMerge(itX->begin, itX->end, itY->end, cmp, params);
79         itX->end = itY->end;
80         runList.erase(itY);
81         appendRun(runList, itX, cmp, params);
82         appendRun(runList, itZ, cmp, params);
83         break;
84     }
85 }
86 };
87
88 // template arguments are copied from http://stackoverflow.com/questions/2447458/
89 template <class RAIterator, class Comparator>
90 void TimSort(RAIterator begin, RAIterator end, Comparator cmp, const timsort::ITimSortParams &params) {
91     using timsort::TRun;
92     using timsort::List;
93     using timsort::insertionSort;
94
95     typedef TRun<RAIterator> Run;
96
97     // step 0: get minimum run length
98     std::size_t length = std::distance(begin, end);
99     std::size_t minrun = params.minRun(length);
100
101     // step 1: split array into runs
102     List<Run> runs;
103
104     // Add dummy runs
105     runs.pushBack(Run(begin, end, 2 * length + 2));
106     runs.pushBack(Run(begin, end, length + 1));
107
108     RAIterator runBegin = begin;
109     while (runBegin != end) {
110         RAIterator runEnd = runBegin + 1;
111         std::size_t runLength = 1;
112         if (runEnd == end) {
113             runs.pushBack(Run(runBegin, runEnd));
114             runBegin = end;
115             break;
116         }
117
118         ++runEnd;
119         ++runLength;
120         bool isIncreasing = !cmp(*(runBegin + 1), *runBegin); // !(B < A) == A <= B,
                                                                // non-strict increasing
121
122         bool currentRunSorted = true;
123         while (runEnd != end && runLength < minrun) {
124             ++runEnd;
125             ++runLength;
126             if (currentRunSorted && !cmp(*(runEnd - 1), *(runEnd - 2)) != isIncreasing) {
127                 currentRunSorted = false;
128             }
129         }
130
131         if (currentRunSorted) {
132             while (runEnd != end && !cmp(*runEnd, *(runEnd - 1)) == isIncreasing) {
133                 ++runEnd;
134                 ++runLength;
135             }
136             if (!isIncreasing) {
137                 timsort::reverse(runBegin, runEnd);
138             }

```

```

139         } else {
140             insertionSort(runBegin, runEnd, cmp);
141         }
142         runs.pushBack(Run(runBegin, runEnd));
143         runBegin = runEnd;
144     }
145
146     // step 2: merging
147     auto runIterator = runs.begin();
148     std::advance(runIterator, 2);
149     while (runIterator != runs.end()) {
150         appendRun(runs, runIterator, cmp, params);
151         ++runIterator;
152     }
153
154     while (runs.size() > 3) {
155         Run rightRun = runs.back();
156         runs.popBack();
157         Run &leftRun = runs.back();
158         inplaceMerge(leftRun.begin, leftRun.end, rightRun.end, cmp, params);
159         leftRun.end = rightRun.end;
160     }
161 }
162
163 template <class RAIterator>
164 void TimSort(RAIterator begin, RAIterator end, const timsort::ITimSortParams &params) {
165     TimSort(begin, end, std::less<typename std::iterator_traits<RAIterator>::value_type>(), params);
166 }
167
168 template <class RAIterator, class Comparator, class =
169     typename std::enable_if<!std::is_base_of<timsort::ITimSortParams, Comparator>::value>::type>
170 void TimSort(RAIterator begin, RAIterator end, Comparator cmp) {
171     TimSort(begin, end, cmp, timsort::StdTimSortParams());
172 }
173
174 template <class RAIterator>
175 void TimSort(RAIterator begin, RAIterator end) {
176     TimSort(begin, end, std::less<typename std::iterator_traits<RAIterator>::value_type>(),
177         timsort::StdTimSortParams());
178 }
179
180 #endif // TIMSORT_TIMSORT_H

```

## 2.3 Оценка времени работы

На первом этапе каждый блок обрабатывается либо за некоторое время, не превосходящее  $\min Run^2(n) \leq C_1$  (время, необходимое для квадратичной сортировки), либо за  $C_2 * k$ , где  $k$  – длина блока,  $C_2$  – время, необходимое для просмотра соответствующих элементов и, возможно, разворачивания массива. Значит, каждый блок набирается за  $Ck$  времени. Значит, весь первый этап проходит за  $Cn \in O(n)$ .

Для оценки времени работы второго этапа воспользуемся методом предоплаты. Пусть на каждый блок в тот момент, когда он впервые начинает рассматриваться, кладётся  $2lh$  монеток, где  $l$  – длина блока,  $h$  – количество блоков перед ним. Пусть также сливание двух блоков в один стоит столько монеток, сколько элементов в сумме в этих двух блоках.

Рассмотрим случаи, когда мы сливаем два блока на втором этапе.

...	X	Y	Z	
-----	---	---	---	--

Если нарушено правило 1, то есть  $Z \geq Y$ , то мы сольём  $Y$  и  $Z$ . При этом мы потратим  $Y + Z$  монеток. Из списка мы можем взять  $2hY + 2(h+1)Z - 2h(Y+Z) = 2Z \geq Z + Y$  монеток. После этого мы рекурсивно запустимся. Значит, в этом случае всё корректно.

Если нарушено правило 2, то есть  $Z + Y \geq X$ , то мы сольём  $X$  и  $Y$ . При этом правило 1 нарушено не было, то есть  $Y > Z$ . Нам нужно  $X + Y$  монеток, мы можем получить  $2hX + 2(h+1)Y + 2(h+2)Z - 2h(X+Y) - 2(h+1)Z = 2Y + 2Z > Y + (Y+Z) \geq Y + X$  монеток. После этого мы рекурсивно запустимся

дважды. Значит, и в этом случае всё корректно.

Докажем, что всего мы положим не больше, чем  $Cn \log_2 n$  монеток. Назовём список *правильным*, если оба свойства в нём выполнены. Тогда  $2Z < Z + Y < X$ . Значит, если первый блок в списке имеет длину  $k$ , то список имеет длину не больше, чем  $2 \log_2 k + 1$ . Перед тем, как добавить очередной блок в список, список правильный. Значит, на этот блок мы кладём не больше, чем  $Ck \log_2 n$  монеток. Значит, всего монеток не больше, чем  $Cn \log_2 n$ . Следовательно, второй этап проходит за  $O(n \log n)$ .

В третьем этапе мы сливаем не больше, чем  $C_1 \log_2 n$  блоков суммарной длиной  $n$ . Значит, все слияния мы выполняем не дольше, чем за  $Cn \log_2 n \in O(n \log n)$  действий.

Таким образом, алгоритм TimSort работает за  $O(n \log n)$ , что и требовалось доказать.