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

Разобъём 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*** *.*

```
#ifndef TIMSORT_TIMSORT_H
#define TIMSORT_TIMSORT_H

#include "list.h"
#include "utils.h"
#include "itertools.h"
#include "merge.h"
#include "inplaceMerge.h"
#include "slowsort.h"
#include "params.h"
#include "debug.h"

#include <iterator>

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

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

```
            }
        } else {
            insertionSort(runBegin, runEnd, cmp);
        }
        runs.pushBack(Run(runBegin, runEnd));
        runBegin = runEnd;
    }

    // step 2: merging
    auto runIterator = runs.begin();
    std::advance(runIterator, 2);
    while (runIterator != runs.end()) {
        appendRun(runs, runIterator, cmp, params);
        ++runIterator;
    }

    while (runs.size() > 3) {
        Run rightRun = runs.back();
        runs.popBack();
        Run &leftRun = runs.back();
        inplaceMerge(leftRun.begin, leftRun.end, rightRun.end, cmp, params);
        leftRun.end = rightRun.end;
    }
}

template <class RAIterator>
void TimSort(RAIterator begin, RAIterator end, const timsort::ITimSortParams &params) {
    TimSort(begin, end, std::less<typename std::iterator_traits<RAIterator>::value_type>(), params);
}

template <class RAIterator, class Comparator, class =
        typename std::enable_if<!std::is_base_of<timsort::ITimSortParams, Comparator>::value>::type>
void TimSort(RAIterator begin, RAIterator end, Comparator cmp) {
    TimSort(begin, end, cmp, timsort::StdTimSortParams());
}

template <class RAIterator>
void TimSort(RAIterator begin, RAIterator end) {
    TimSort(begin, end, std::less<typename std::iterator_traits<RAIterator>::value_type>(),
            timsort::StdTimSortParams());
}

#endif // TIMSORT_TIMSORT_H
```

2.3 Оценка времени работы

На первом этапе каждый блок обрабатывается либо за некоторое время, не превосходящее $minRun^2(n) \leq C_1$ (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.