TimSort

Владимир Огородников

26 октября 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 возратающем 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++

```
#include <list>
#include <iterator>
#include <algorithm>
#include <stdexcept>

namespace timsort {
    template <class RAIterator>
    struct TRun {
        RAIterator begin, end;
        std::size_t dummyLength;
        TRun(RAIterator begin, RAIterator end, std::size_t dummyLength = 0) : begin(begin),
                end(end), dummyLength(dummyLength) {  }
        std::size_t length() const {
            return dummyLength ? dummyLength : std::distance(begin, end);
        }
        bool dummy() const {
            return dummyLength; // != 0
        }
```

```
};

std::size_t getMinrun(std::size_t length) {
    std::size_t r = 0;
    while (length >= 64) {
        r |= length & 1;
        length >>= 1;
    }
    return length + r;
}

double abs(double x) {
    return x < 0 ? -x : x;
}

unsigned long long sqrt(unsigned long long n) {
    double a = 1e6; // Just a random initial value
    while (abs(a * a - n) > 1) {
        a = (a + n / a) / 2;
    }
    unsigned long long ans = a + 3;
    while (ans * ans > n) {
        --ans;
    }
    return ans;
}

template <class RAIterator>
void swapBlocks(RAIterator leftBlock, RAIterator rightBlock, std::size_t length) {
    while (length--) {
        std::swap(*leftBlock++, *rightBlock++);
    }
}

template <class RAIterator, class Comparator>
struct Block {
    RAIterator begin;
    std::size_t length;
    Comparator cmp;
    Block(RAIterator begin, std::size_t length, Comparator cmp) : begin(begin), length(length),
            cmp(cmp) {}
    void swap(const Block<RAIterator, Comparator> &other) const {
        swapBlocks(begin, other.begin, length);
    }
    bool operator <(const Block<RAIterator, Comparator> &other) const {
        bool c1 = cmp(*begin, *other.begin);
        if (!c1) {
            return !cmp(*other.begin, *begin) && cmp(*(begin + length - 1),
                    *(other.begin + other.length - 1));
        }
        return c1;
    }

};

template <class RAIterator, class Comparator>
void swap(Block<RAIterator, Comparator> &b1, Block<RAIterator, Comparator> &b2) {
    b1.swap(b2);
}

template <class RAIterator, class Comparator>
```

```
80      class BlockIterator {
81      private:
82          Block<RAIterator, Comparator> curBlock;
83      public:
84          typedef typename std::iterator_traits<RAIterator>::value_type value_type;
85          typedef typename std::iterator_traits<RAIterator>::reference reference;
86          typedef typename std::iterator_traits<RAIterator>::pointer pointer;
87          typedef typename std::iterator_traits<RAIterator>::difference_type difference_type;
88          typedef std::random_access_iterator_tag iterator_category;
89
90          BlockIterator(const Block<RAIterator, Comparator> block) : curBlock(block) {}
91          Block<RAIterator, Comparator> &operator *() {
92              return curBlock;
93          }
94          bool operator !=(const BlockIterator<RAIterator, Comparator> &iter) const {
95              return curBlock.begin != iter.curBlock.begin;
96          }
97          BlockIterator<RAIterator, Comparator> &operator ++() {
98              curBlock.begin += curBlock.length;
99              return *this;
100         }
101         BlockIterator<RAIterator, Comparator> operator ++(int) {
102             auto copy = *this;
103             ++(*this);
104             return copy;
105         }
106         BlockIterator<RAIterator, Comparator> operator +=(std::size_t n) {
107             curBlock.begin += n * curBlock.length;
108             return *this;
109         }
110         const BlockIterator<RAIterator, Comparator> operator +(std::size_t n) {
111             BlockIterator<RAIterator, Comparator> ans = *this;
112             ans += n;
113             return ans;
114         }
115     };
116
117     template <class RAIterator, class T = typename std::iterator_traits<RAIterator>::value_type,
118         class Comparator = std::less<T>>
119     void selectionSort(RAIterator begin, RAIterator end, Comparator cmp = Comparator()) {
120         using std::swap;
121         RAIterator unsortedBegin = begin;
122         while (unsortedBegin != end) {
123             RAIterator minElement = unsortedBegin;
124             for (RAIterator iter = unsortedBegin + 1; iter != end; ++iter) {
125                 if (cmp(*iter, *minElement)) {
126                     minElement = iter;
127                 }
128             }
129             swap(*unsortedBegin++, *minElement);
130         }
131     }
132
133     template <class RAIterator, class T = typename std::iterator_traits<RAIterator>::value_type,
134         class Comparator = std::less<T>>
135     void insertionSort(RAIterator begin, RAIterator end, Comparator cmp = Comparator()) {
136         using std::swap;
137         for (RAIterator i = begin; i != end; ++i) {
138             RAIterator j = i;
139             while (j != begin && cmp(*j, *(j - 1))) {
140                 swap(*j, *(j - 1));
```

```
                --j;
            }
        }
    }

    template <class RAIterator, class Comparator>
    void mergeWithBuffer(RAIterator leftBegin, RAIterator leftEnd,
            RAIterator rightBegin, RAIterator rightEnd,
            RAIterator bufBegin, Comparator cmp) {
        std::size_t leftLength = std::distance(leftBegin, leftEnd);
        swapBlocks(leftBegin, bufBegin, leftLength);
        RAIterator bufEnd = bufBegin + leftLength;
        RAIterator it1 = bufBegin, it2 = rightBegin, itAns = leftBegin;
        while (it1 != bufEnd && it2 != rightEnd) {
            if (!cmp(*it2, *it1)) {
                std::swap(*it1++, *itAns++);
            } else {
                std::swap(*it2++, *itAns++);
            }
            if (itAns == leftEnd) {
                itAns = rightBegin;
            }
        }

        while (it1 != bufEnd) {
            std::swap(*it1++, *itAns++);
            if (itAns == leftEnd) {
                itAns = rightBegin;
            }
        }

        while (it2 != rightEnd) {
            std::swap(*it2++, *itAns++);
            if (itAns == leftEnd) {
                itAns = rightBegin;
            }
        }
    }

    template <class RAIterator>
    class IteratorReverser {
    private:
        RAIterator iter_;
    public:
        typedef typename std::iterator_traits<RAIterator>::value_type value_type;
        typedef typename std::iterator_traits<RAIterator>::reference reference;
        typedef typename std::iterator_traits<RAIterator>::pointer pointer;
        typedef typename std::iterator_traits<RAIterator>::difference_type difference_type;
        typedef std::random_access_iterator_tag iterator_category;

        IteratorReverser(const RAIterator &it) : iter_(it) {}
        value_type &operator *() const {
            return *(iter_ - 1);
        }
        const IteratorReverser<RAIterator> operator ++(int) {
            IteratorReverser<RAIterator> ans = *this;
            --iter_;
            return ans;
        }
        const IteratorReverser<RAIterator> operator +(const difference_type diff) const {
            return IteratorReverser<RAIterator>(iter_ - diff);
```

```
202         }
203         bool operator ==(const IteratorReverser<RAIterator> &other) {
204             return iter_ == other.iter_;
205         }
206         bool operator !=(const IteratorReverser<RAIterator> &other) {
207             return !operator ==(other);
208         }
209         difference_type operator -(const IteratorReverser<RAIterator> &other) const {
210             return other.iter_ - iter_;
211         }
212     };
213 
214     template <class RAIterator>
215     const IteratorReverser<RAIterator> reverseIter(const RAIterator &iter) {
216         return IteratorReverser<RAIterator>(iter);
217     }
218 
219     template <class Comparator, class T>
220     class InvertedComparator {
221     private:
222         Comparator cmp_;
223     public:
224         InvertedComparator(const Comparator &cmp) : cmp_(cmp) {}
225         bool operator() (const T &a, const T &b) {
226             return cmp_(b, a);
227         }
228     };
229 
230     template <class RAIterator, class Comparator,
231               class T = typename std::iterator_traits<RAIterator>::value_type>
232     void backwardMergeWithBuffer(RAIterator leftBegin, RAIterator leftEnd,
233             RAIterator rightBegin, RAIterator rightEnd,
234             RAIterator bufBegin, Comparator cmp) {
235         double length = std::distance(rightBegin, rightEnd);
236         mergeWithBuffer(reverseIter(rightEnd), reverseIter(rightBegin),
237                 reverseIter(leftEnd), reverseIter(leftBegin),
238                 reverseIter(bufBegin + length), InvertedComparator<Comparator, T>(cmp));
239     }
240 
241     template <class RAIterator, class Comparator>
242     void inplaceMerge(const TRun<RAIterator> &runLeft, const TRun<RAIterator> &runRight,
243             Comparator cmp) {
244         if (runLeft.dummy() || runRight.dummy() || runLeft.end != runRight.begin) {
245             throw std::runtime_error("Incorrect inplace merge");
246         }
247         RAIterator begin = runLeft.begin, end = runRight.end;
248         std::size_t length = std::distance(begin, end);
249 
250         if (length < 4) {
251             selectionSort(begin, end, cmp);
252             return;
253         }
254 
255         std::size_t blockLength = sqrt(length);
256         std::size_t blocksCount = length / blockLength;
257         std::size_t secondRun = runLeft.length();
258         std::size_t blockWithFirstRunEnd = (secondRun - 1) / blockLength;
259         RAIterator lastBlockBegin = begin + (blocksCount - 1) * blockLength;
260 
261         swapBlocks(begin + blockWithFirstRunEnd * blockLength, lastBlockBegin, blockLength);
262 
```

```
263         selectionSort(BlockIterator<RAIterator, Comparator>(
264                     Block<RAIterator, Comparator>(begin, blockLength, cmp)),
265                 BlockIterator<RAIterator, Comparator>(
266                     Block<RAIterator, Comparator>(lastBlockBegin, blockLength, cmp)),
267                 std::less<Block<RAIterator, Comparator>>());
268 
269         for (std::size_t i = 0; i < blocksCount - 2; ++i) {
270             // Merge i & i+1 blocks using last block as buffer
271             RAIterator leftBlockBegin = begin + i * blockLength;
272             RAIterator rightBlockBegin = leftBlockBegin + blockLength;
273 
274             mergeWithBuffer(leftBlockBegin, leftBlockBegin + blockLength,
275                     rightBlockBegin, rightBlockBegin + blockLength, lastBlockBegin, cmp);
276         }
277 
278         std::size_t lastBigBlockLength = length - (blocksCount - 1) * blockLength;
279         RAIterator doubleBigBlockBegin = begin + length - 2 * lastBigBlockLength;
280         selectionSort(doubleBigBlockBegin, end, cmp);
281 
282         backwardMergeWithBuffer(begin, doubleBigBlockBegin,
283                 doubleBigBlockBegin, doubleBigBlockBegin + lastBigBlockLength,
284                 doubleBigBlockBegin + lastBigBlockLength, cmp);
285         selectionSort(doubleBigBlockBegin + lastBigBlockLength, end, cmp);
286     }
287 
288     template <class RAIterator, class Comparator>
289     void appendRun(std::list<TRun<RAIterator>> &runList,
290             typename std::list<TRun<RAIterator>>::iterator newRun, Comparator cmp) {
291         auto itZ = newRun, itY = newRun, itX = newRun;
292         std::advance(itY, -1);
293         std::advance(itX, -2);
294         if (itZ->length() >= itY->length()) {
295             inplaceMerge(*itY, *itZ, cmp);
296             itZ->begin = itY->begin;
297             runList.erase(itY);
298             appendRun(runList, itZ, cmp);
299         } else if (itZ->length() + itY->length() > itX->length()) {
300             inplaceMerge(*itX, *itY, cmp);
301             itX->end = itY->end;
302             runList.erase(itY);
303             appendRun(runList, itX, cmp);
304             appendRun(runList, itZ, cmp);
305         }
306     }
307 };
308 
309 // template arguments are copied from http://stackoverflow.com/questions/2447458/
310 template <class RAIterator, class Comparator = std::less<
311     typename std::iterator_traits<RAIterator>::value_type> >
312 void TimSort(RAIterator begin, RAIterator end, Comparator cmp = Comparator()) {
313     using namespace timsort;
314     typedef TRun<RAIterator> Run;
315 
316     // step 0: get minimum run length
317     std::size_t length = std::distance(begin, end);
318     std::size_t minrun = getMinrun(length);
319 
320     // step 1: split array into runs
321     std::list<Run> runs;
322 
323     // Add dummy runs
```

```
    runs.push_back(Run(begin, end, 2 * length + 2));
    runs.push_back(Run(begin, end, length + 1));

    RAIterator runBegin = begin;
    while (runBegin != end) {
        RAIterator runEnd = runBegin + 1;
        std::size_t runLength = 1;
        if (runEnd == end) {
            runs.push_back(Run(runBegin, runEnd));
            runBegin = end;
            break;
        }

        ++runEnd;
        ++runLength;
        bool isIncreasing = !cmp(*(runBegin + 1), *runBegin); // !(B < A) == A <= B,
                                                              // non-strict increasing
        bool currentRunSorted = true;
        while (runEnd != end && runLength < minrun) {
            ++runEnd;
            ++runLength;
            if (currentRunSorted && cmp(*(runEnd - 2), *(runEnd - 1)) != isIncreasing) {
                currentRunSorted = false;
            }
        }

        if (currentRunSorted) {
            while (runEnd != end && cmp(*(runEnd - 1), *runEnd) == isIncreasing) {
                ++runEnd;
                ++runLength;
            }
            if (!isIncreasing) {
                std::reverse(runBegin, runEnd);
            }
        } else {
            insertionSort(runBegin, runEnd, cmp);
        }
        runs.push_back(Run(runBegin, runEnd));
        runBegin = runEnd;
    }

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

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

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.