Merge Strategies: from Merge Sort to TimSort

Nicolas Auger, Cyril Nicaud, Carine Pivoteau

Merge Strategies: from Merge Sort to TimSort

Nicolas Auger, Cyril Nicaud, and Carine Pivoteau

Université Paris-Est, LIGM (UMR 8049), F77454 Marne-la-Vallée, France
{auger,nicaud,pivoteau}@univ-mlv.fr

Abstract. The introduction of TimSort as the standard algorithm for sorting in Java and Python questions the generally accepted idea that merge algorithms are not competitive for sorting in practice. In an attempt to better understand TimSort algorithm, we define a framework to study the merging cost of sorting algorithms that relies on merges of monotonic subsequences of the input. We design a simpler yet competitive algorithm in the spirit of TimSort based on the same kind of ideas. As a benefit, our framework allows to establish the announced running time of TimSort, that is, $\mathcal{O}(n \log n)$.

1 Introduction

TimSort [4] is a sorting algorithm designed in 2002 by Tim Peters, for use in the Python programming language. It was thereafter implemented in other well-known programming languages such as Java. It is quite a strongly engineered algorithm, but its high-level principle is rather simple: The sequence S to be sorted is decomposed into monotonic runs (*i.e.*, nonincreasing or nondecreasing subsequences of S), which are merged pairwise according to some specific rules.

In order to understand and analyze the merging strategy (meaning the order in which the merges are performed) of TimSort, we consider a whole class of sorting algorithms that relies on a stack to decide in which in order the merges are performed. Other ways to order merges have been considered in the litterature: the classical MergeSort is one of them, but also Knuth's NaturalMergeSort [3] and the optimal merging strategy proposed by Barbay and Navarro [1], using the ideas of Huffmann incoding.

Our framework consists in adding the runs one by one in a stack, and in performing merges on the go according some rules. This rules are always local as they only involve the runs at the top of the stack. Following this general framework, we propose a simpler variant of TimSort. Finally, we establish that both TimSort and this new algorithm run in $\mathcal{O}(n \log n)$ time.[1]

2 Settings

2.1 Sequences and runs

For every positive integers i and j, let $[i] = \{1, \ldots, i\}$ and let $[i, j] = \{i, \ldots, j\}$. Let $(E, \leq)$ be a totally ordered set. In this article, we consider non-empty finite

[1] This fact is a folklore result for TimSort, but it does not seem to appear anywhere.

Algorithm 1: Generic Run-Merge Sort for S

```
1 R ← run decomposition of S
2 while |R| ≠ 1 do
3 |   remove two runs R and R' of R
4 |   add merge(R, R') to R
5 if the unique run R_1 in R is nonincreasing then reverse R_1
6 return R_1
```

sequences of elements of E, that is, elements of $E^+ = \cup_{n\geq 1} E^n$. The *length* $|S|$ of such a sequence is its number of elements. A sequence $S = (s_1, \ldots, s_n)$ is *sorted* when, for every $i \in [n-1]$, $s_i \leq s_{i+1}$. We are interested in *sorting* algorithms that, for any given sequence $S \in E^n$, find a permutation σ of $[n]$ such that $(s_{\sigma(1)}, \ldots, s_{\sigma(n)})$ is sorted. Most of the time, we do not want σ explicitly, but instead directly compute the sequence $\mathbf{sort}(S) = (s_{\sigma(1)}, \ldots, s_{\sigma(n)})$.

A *run* of a sequence $S = (s_1, \ldots, s_n)$ is a non-empty subsequence $(s_i, \ldots, s_j)$ such that either $(s_i, \ldots, s_j)$ or $(s_j, \ldots, s_i)$ is sorted. The former is a *nondecreasing run*, and the latter is a *nonincreasing run*.[2] A *run decomposition* of a sequence S of length n is a nonempty sequence $\mathcal{R} = (R_1, \ldots, R_m)$ of elements of E^+ such that each R_i is a run (either nondecreasing or nonincreasing), and such that $S = R_1 \cdot R_2 \cdots R_m$, where $R \cdot R'$ denote the classical concatenation of sequences.

Example 1. $\mathcal{R}_1 = (2,3,5,7,11) \cdot (10) \cdot (9) \cdot (8,9,10)$ and $\mathcal{R}_2 = (2,3,5,7,11) \cdot (10,9,8) \cdot (9,10)$ are two run decompositions of $S = (2,3,5,7,11,10,9,8,9,10)$.

2.2 Run-merge sorting algorithms and run decomposition strategies

We now equip the runs with a merge operation. If R and R' are two runs, let $\mathbf{merge}(R, R')$ denote the sequence made of the elements of R and R' placed in nondecreasing order, *i.e.*, $\mathbf{merge}(R, R') = \mathbf{sort}(R \cdot R')$.

In this article we are interested in sorting algorithms that follow what we call a *generic run-merge sort* template. Such algorithms consist of two steps: First the sequence is split into a run decomposition. Then, the runs are merged pairwise until only one remains, which is the sorted sequence associated with the input.[3] This generic algorithm is depicted in Algorithm 1.

To design such an algorithm, the two main concerns are how the run decomposition is computed, and in which order the runs are merged. Observe that several classical algorithms fit in this abstract settings.

MergeSort is a run-merge sorting algorithm in which each run is reduced to a single element. Note that, in this case, the cost of computing the run decomposition is $O(1)$. Then, the runs are merged according to the recursive calls made during the divide and conquer algorithm.

[2] Observe that we do not require a run to be maximal, though they will usually be.
[3] Except in the very specific case where $\mathcal{R}$ consists of only one nonincreasing run.

NaturalMergeSort is a variation of merge sort proposed by Knuth [3]. It consists in first decomposing the sequence into maximal nondecreasing runs, then in using the same merging strategy as in MergeSort. The run decomposition can be obtained by scanning the input sequence S from left to right and by starting a new run every time an element is smaller than the one before. This uses $n-1$ comparisons for a sequence of length n.

TimSort [4] is a relatively new algorithm that is implemented in standard libraries of several common programming languages (Python, Java, ...). This algorithm is highly engineered and uses many efficient heuristics, but this is not our purpose to fully describe it here. However, we are quite interested in the run-merging strategy it relies on, which consists in first computing a decomposition into maximal nonincreasing and nondecreasing runs, which are then merged using a stack. The merging strategy is defined by some invariants that must be satisfied within this stack (merges occur when they are not) and we will give more details on this in Section 3.

Concerning the run decomposition, the idea is to take advantage of the natural maximal runs of S, but each run can be either nonincreasing or nondecreasing, according to order of its first two elements.[4] As for the previous solution, $n-1$ comparisons are required to calculate this decomposition. In *Example 1*, $\mathcal{R}_1$ was computed as in NaturalMergeSort and $\mathcal{R}_2$ as in TimSort.

Since the number of useful strategies to compute a run decomposition is limited, we choose to mostly focus on merging strategies in this paper.

2.3 Merging cost and optimal merging strategy

We now turn our attention to the cost of a merge and we consider that the sequence S is stored in an array A of size n (and the runs are encoded by their starting and ending indices in A). The classical implementations of the merging procedure use an auxiliary array of length $\min(|R|, |R'|)$, where the smallest run is copied.[5] Then, the number of comparisons is at most $|R| + |R'| - 1$.

In the sequel, we therefore consider that the number of comparisons needed to merge two runs R and R' is $\mathbf{c}(R, R') - 1$, where $\mathbf{c}(R, R') = |R| + |R'|$.

If we only consider this cost function, the optimal merging strategy is given in by Barbay and Navarro [1]. However, in practice, among sorting strategies based on merging runs, TimSort is prefered to other sorting algorithms: The hidden cost for computing the optimal strategy, as well as many other parameters such as cache misses, are involved in the actual running time.

We therefore focus on strategies a la TimSort in the following, and we propose a generic framework to design sorting algorithms that benefit from the same features than TimSort.

[4] Actually, in TimSort, the size of short runs is artificially increased. We do not consider this feature here and focus on the basic ideas of TimSort only.

[5] This extra memory requirement is a reason why QuickSort is sometimes preferred to MergeSort, even if it performs more comparisons in the worst and average cases.

Algorithm 2: Generic Stack Run-Merge Sort for the strategy $\mathfrak{S}$

```
1  R ← run decomposition of S
2  X ← ∅
3  while R ≠ ∅ do
4      R ← pop(R)
5      Append R to X
6      while X violates at least one rule of 𝔖 do
7          (ρ, μ) ← first pair such that ρ is not satisfied
8          Apply μ to X                                  /* ρ is activated */
9  while |X| ≥ 1 do
10     R, R' ← pop(X), pop(X)
11     Append merge(R, R') to X
12 return the unique element of X
```

3 TimSort-like strategies

As mentioned above, TimSort is a recent sorting algorithm for arrays that follows our generic run-merge template. It contains many heuristics that will not be discussed here. In the sequel, we will therefore focus on analyzing the merge strategy used by TimSort. We will not consider how the merges are actually performed. They are many heuristics involved in the process, but they do not alter the worst case analysis.

Before describing TimSort in details, we propose a framework to design a whole class of merge strategies based on the use of a stack, which we call *stack strategies*. TimSort will be an example of such a strategy.

3.1 Stack strategies

Let $\mathcal{R} = (R_1, \ldots, R_m)$ be a run decomposition. A stack strategy relies on a stack $\mathcal{X}$ of runs that is initially empty. During the first stage, at each step, a run is extracted from $\mathcal{R}$ and added to the stack. The stack is then updated, by merging runs, in order to assure that some conditions on the top of the stack are satisfied. These conditions and the way runs are merged when they are not satisfied define the strategy. The second stage occurs when there is no more run in $\mathcal{R}$: the runs in $\mathcal{X}$ are then merged pairwise until only one remains.

A *rule* of degree $k \geq 2$ is a property of a stack $\mathcal{X}$ that involves the k topmost elements of $\mathcal{X}$. By convention, the rule is always satisfied when there are less than k elements in $\mathcal{X}$. A *merge strategy* of degree k is the process of merging some of the k topmost runs in a stack $\mathcal{X}$; at least one merge must be performed. A *stack strategy* of degree k consists of a nonempty sequence of s pairs (ρ_i, μ_i), where ρ_i is a rule of degree at most k and μ_i is a merge strategy of degree at most k. The *stack-merge algorithm* associated with the stack strategy $\mathfrak{S} = \langle(\rho_1, \mu_1), \ldots, (\rho_s, \mu_s)\rangle$ is depicted in Algorithm 2. The order of the rules matters:

Algorithm 3: `merge_collapse(stack ms)`

```
while (n > 1) {
  n = size(ms) - 2;
  if ((n > 0 && p[n-1].len <= p[n].len + p[n+1].len) ||
     (n > 1 && p[n-2].len <= p[n-1].len + p[n].len)) {
     if (p[n-1].len < p[n+1].len)
       --n;
     if (merge_at(ms, n) < 0)
       return -1;
  }
  else
  if (p[n].len <= p[n+1].len) {
       if (merge_at(ms, n) < 0)
         return -1;
  }
  else break;
}
return 0;
```

when one or several rules are violated, the merge strategy associated with the first one, and only this one, is performed. This rule is said to be *activated*.

Note that such an algorithm always halts, as the inner loop reduces the number of runs in $\mathcal{X}$: at some point the stack $\mathcal{X}$ contains less elements than the minimal degree of the rules, which are then all satisfied.

3.2 TimSort and α-StackSort

TimSort can be seen as a stack-merge algorithm of degree 4. It is not the way it is usually written, but it is strictly equivalent to the following merge strategy, for a stack that ends with the runs W, X, Y and Z:

- $\rho_1 := |X| \geq |Z|$ and μ_1 consists in merging X and Y;
- $\rho_2 := |X| > |Y| + |Z|$ and μ_2 consists in merging Y and Z;
- $\rho_3 := |W| > |X| + |Y|$ and μ_3 consists in merging Y and Z;
- $\rho_4 := |Y| > |Z|$ and μ_4 consists in merging Y and Z;

An example of the successive states of the stack is given in Fig 1. Remark that in the original version of TimSort, the third rule was missing. This lead to

	ρ_4 violated			ρ_2 violated	ρ_4 violated	
•	•	•	•	(8, 9)	•	•
•	(6, 5, 2)	•	(3, 4, 10)	(3, 4, 10)	(3, 4, 8, 9, 10)	•
(1, 7)	(1, 7)	(1, 2, 5, 6, 7)	(1, 2, 5, 6, 7)	(1, 2, 5, 6, 7)	(1, 2, 5, 6, 7)	(1, . . . , 10)

Fig. 1. The stack configurations during the execution of TimSort for the sequence $\mathcal{S} = (1, 7, 6, 5, 2, 3, 4, 10, 8, 9)$.

Algorithm 4: merge_collapse translated

```
1  R ← run decomposition of S
2  X ← ∅
3  while R ≠ ∅ do
4      R ← pop(R)
5      Append R to X
6      while True do
7          if |X| ≤ |Y| + |Z| or |W| ≤ |X| + |Y| then
8              if |X| < |Z| then  Merge X and Y
9              else  Merge Y and Z
10         else if |Y| ≤ |Z| then
11             Merge Y and Z
12         else
13             break
```

some problems; in particular, Lemma 2 did not hold without this rule. This new rule was proposed in [2], where the problem in the invariant was first identified, and quickly corrected in Python.[6]

Algorithm 3 is the code (in C) found in the Python library for the main loop of TimSort. We "translate" it into pseudo-code, using the convention that $\mathcal{X} = (x_1, \cdots, x_{\ell-4}, W, X, Y, Z)$. Hence W, X, Y, and Z are the four topmost elements of the stack, Z being on top. The result is given in Algorithm 4.

Important: whenever a predicate in a `if` conditional uses a variable that is not available because the stack is too small, for instance $|W| \leq |X| + |Y|$ for a stack of size 3, we consider that this predicate is false. This convention is useful to avoid many tests on the length of the stack.

As Algorithm 4 is not exactly in the form of pairs (rules,merge), we rewrite the algorithm as depicted in Algorithm 5. The rules and merge strategies are, in order:

$$\begin{array}{ll} \rho_1 := |X| \geq |Z| & \mu_1 = \mathbf{merge}(X, Y) \\ \rho_2 := |X| > |Y| + |Z| & \mu_2 = \mathbf{merge}(Y, Z) \\ \rho_3 := |W| > |X| + |Y| & \mu_3 = \mathbf{merge}(Y, Z) \\ \rho_4 := |Y| > |Z| & \mu_4 = \mathbf{merge}(Y, Z) \end{array}$$

We propose our own stack-merge algorithm α-StackSort, which is of degree 2. It depends on a fixed parameter $\alpha > 1$, and consists only in one rule ρ which is $|Y| > \alpha\,|Z|$. If it is violated, μ consists in merging Y and Z. The algorithm α-StackSort is therefore a very simple stack-merge algorithm.

[6] `https://hg.python.org/cpython/file/default/Objects/listobject.c`

Algorithm 5: TimSort (S, n)

```
1  R ← run decomposition of S
2  X ← ∅
3  while R ≠ ∅ do
4      R ← pop(R)
5      Append R to X
6      while X violates at least one rule of 𝔖 do
7          if |X| < |Z| then                          /* ρ1 is activated */
8              Merge X and Y
9          else if |X| ≤ |Y| + |Z| then               /* ρ2 is activated */
10             Merge Y and Z
11         else if |W| ≤ |X| + |Y| then               /* ρ3 is activated */
12             Merge Y and Z
13         else if |Y| ≤ |Z| then                     /* ρ4 is activated */
14             Merge Y and Z
```

4 Analysis of TimSort and α-StackSort

The rules and merge strategies of both algorithms are designed in order to ensure that some global invariants hold throughout the stack. This is the meaning of the next two lemmas. The part on TimSort was proven in [2].

Lemma 1. *Algorithm 4 and Algorithm 5 are equivalent: for given sequence of runs $\mathcal{R}$, they are both characterized by the same merge tree.*

Proof. Straightforward. Just check that the conditions that leads to the merge of X and Y, Y and Z or no merge are equivalent. □

Lemma 2. *Let $\mathcal{X} = (x_1, \ldots, x_\ell)$ be the stack configuration at the beginning of any iteration of the while loop at line 3 of Algorithm 2.*
- *For* TimSort *we have $|x_i| > |x_{i+1}| + |x_{i+2}|$, for every $i \in [\ell - 2]$.*
- *For α-*StackSort *we have $|x_i| > \alpha\,|x_{i+1}|$, for every $i \in [\ell - 1]$.*

Proof. As already stated, the case of TimSort was proved in [7].

Let us consider α-StackSort. The proof is done by induction on the iteration t of the while loop. The result holds trivially for $t = 1$, since at the first iteration, the stack is empty. Assume it holds at iteration t and consider iteration $t + 1$. During the $(t + 1)$-th iteration, we append a new run R at the end of the stack. If $|x_\ell| > \alpha\,|R|$ then we are done. Otherwise, the inner while loop merge the two rightmost runs Y and Z until they verify $|Y| > \alpha|Z|$. As the condition is satisfied everywhere before by induction hypothesis, this ensures that the condition is satisfied everywhere. □

As remarked by Tim Peters who designed TimSort, the stack contains $\mathcal{O}(\log n)$ runs at any time. This is the same for α-StackSort.

Lemma 3. *At any time during the execution of* TimSort *or* α-StackSort *on a sequence of length n, the stack $\mathcal{X}$ contains $\mathcal{O}(\log n)$ runs.*

Proof. It is sufficient to prove the lemma at the beginning of each iteration of the while loop of line 3 in Algorithm 2, as the stack can have at most one more element, when it is inserted at line 5.

For TimSort, Lemma 4 ensures that if the stack is $\mathcal{X} = (x_1, \cdots, x_\ell)$, then $|x_i| > |x_{i+1}| + |x_{i+2}|$ for every $i \in [\ell - 2]$. Moreover $|x_{\ell-1}| > |x_\ell|$. Hence, by direct induction, for every $i \in [\ell]$, $|x_{\ell-i+1}| \geq |x_\ell|\, F_i$, where F_i is the i-th Fibonacci number. As $F_n \geq c\phi^n$ for $n \geq 1$ and some well chosen c, we have

$$\sum_{i=1}^{\ell} |x_i| \geq |x_\ell| \sum_{i=1}^{\ell} c\phi^i \geq c\,\frac{\phi^\ell - 1}{\phi - 1} = \Omega(\phi^\ell).$$

Since the sum of the run lengths is at most n, we get that $\ell = \mathcal{O}(\log n)$.

The proof for α-StackSort is similar, α playing the role of ϕ.

Next theorem is a folklore result for TimSort, announced in the first description of the algorithm [4]. However, we could not find its proof anywhere in the literature. The same result holds for α-StackSort. Notice that this is not a direct consequence of Lemma 3: if we merge the runs as they arrive, the stack has size $\mathcal{O}(1)$ but the running time is $\mathcal{O}(n^2)$.

Theorem 1. *The number of comparisons needed for* TimSort *and* α-StackSort *to sort a sequence of length n is $\mathcal{O}(n \log n)$.*

Proof of Theorem 1 for α-StackSort

We start with α-StackSort. The run decomposition uses only $n - 1$ comparisons. To analyze the complexity of the while loop of line 3, we rely on a classical technique used for amortized complexity and rewrite this part of the algorithm as in Algorithm 6. In blue have been added some computation on a variable C. Observe first that C is decreased at line 9 every time a merge is done, by an amount equal to the cost of this merge.

We now prove that after each blue instruction (Line 2, Line 6 and Line 9), we have, if the current stack is $\mathcal{X} = (x_1, \ldots, x_\ell)$,

$$C \geq \sum_{i=1}^{\ell} (1 + \alpha) i |x_i| \tag{1}$$

We prove this property by induction: It clearly holds after Line 2. Observe that any time the stack is altered, C is updated immediately after. Hence we just have to prove that if the property holds before an alteration, then it still holds when C is updated just after:

- For Lines 5-6: if $\mathcal{X} = (x_1, \ldots, x_\ell)$ before Line 5, then $\mathcal{X} = (x_1, \ldots, x_\ell, R)$ after Line 6. By induction hypothesis $C \geq \sum_{i=1}^{\ell} (1 + \alpha) i |x_i|$ before Line 5, and it is increased by $(1 + \alpha)(\ell + 1)|R|$ at Line 6. Therefore the property still holds after Line 6.

Algorithm 6: Main loop of α-StackSort

```
1 X ← ∅
2 C ← 0
3 while R ≠ ∅ do
4     R ← pop(R)
5     Append R to X
6     C ← C + (1 + α) |X| |R|                    /* used for the proof only */
7     while X violates the rule |Y| ≥ α |Z| do
8         Merge Y and Z
9         C ← C − (|Y| + |Z| − 1)                /* used for the proof only */
```

– For Lines 8-9: if $\mathcal{X} = (x_1, \dots, x_{\ell-2}, Y, Z)$ before Line 8, then after Line 9 the stack is

$$\mathcal{X} = (x_1, \dots, x_{\ell-2}, \mathbf{merge}(Y, Z)).$$

By induction hypothesis, before Line 8 we have

$$\begin{aligned} C &\geq \sum_{i=1}^{\ell-2}(1+\alpha)i|x_i| + (1+\alpha)(\ell-1)|Y| + (1+\alpha)\ell|Z| \\ &\geq \sum_{i=1}^{\ell-2}(1+\alpha)i|x_i| + (1+\alpha)(\ell-1)(|Y|+|Z|) + (1+\alpha)|Z|. \end{aligned}$$

But we are in the case where the rule is activated, hence $|Y| < \alpha\,|Z|$. Thus $(1+\alpha)|Z| > |Y| + |Z| > |Y| + |Z| - 1$. This yields

$$C - (|Y| + |Z| - 1) \geq \sum_{i=1}^{\ell-2}(1+\alpha)i|x_i| + (1+\alpha)(\ell-1)(|Y|+|Z|).$$

Hence, the property still holds after Line 9.

The quantity C is increased on Line 6 only. By Lemma 5, when a new run R is added in $\mathcal{X}$, C is increased by at most $K \log n\,|R|$, for some positive constant K. Hence, the sum of all increases of C is bounded from above by $K \log n \sum_{R\in\mathcal{R}} |R| = \mathcal{O}(n \log n)$. The quantity C is decreased whenever a merge is performed, by an amount equal to this merge cost. As we just proved that Equation (1) always holds after an update of C, C is non-negative at the end of this part of the algorithm. Hence, the total number of comparisons performed in this part is $\mathcal{O}(n \log n)$.

The last while loop of Algorithm 2 also performs at most $\mathcal{O}(n \log n)$ comparisons, as the stack is of length $\mathcal{O}(\log n)$: every run is involved in at most $\mathcal{O}(\log n)$ merges during this loop. □

Proof of Theorem 1 for TimSort

We want to proceed for TimSort as for α-StackSort, but there are some technical difficulties inherent to the structure of the rules in TimSort. We still

define a variable C initialized with 0 and which is increased by $3iR$ whenever a run R arrive at position i on the stack. We still remove $|R| + |R'| - 1$ from C whenever R and R' are merged. However, we cannot directly guarantee that C is always positive; for some cases we need to consider several consecutive merges made by the algorithm in order to conclude. Hence, we will unroll the main while loop as needed, to obtain an algorithm equivalent to the main while loop, but that is bigger. On this redundant code we then prove that at the end of any iteration, if the stack is $\mathcal{X} = (x_1, \ldots, x_\ell)$ then

$$C \geq \sum_{i=1}^{\ell} 3i|x_i|. \tag{2}$$

We first establish three lemmas, which give hints of what happens, in certain cases, during two consecutive iterations of the while loop.

Lemma 4. *If rule ρ_2 is activated, then rule ρ_4 is violated at the next iteration of the while loop.*

Proof. If rule ρ_2 is activated, then the runs Y and Z are merged. The new stack is $\mathcal{X}' = (x_1, \ldots, x_{\ell-3}, Y', Z')$ with $Y' = X$ and $Z' = \mathbf{merge}(Y, Z)$. Since ρ_2 is violated, we have $|X| \leq |Y| + |Z|$, thus $|Y'| \leq |Z'|$ which is the negation of ρ_4. □

Lemma 5. *If rule ρ_3 is activated, then the rule ρ_2 is violated at the next iteration.*

Proof. If rule ρ_3 is activated, then the runs Y and Z are merged. The new stack is $\mathcal{X}' = (x_1, \ldots, x_{\ell-4}, X', Y', Z')$ with $X' = W$, $Y' = X$, and $Z' = \mathbf{merge}(Y, Z)$. Since ρ_3 is violated, we have $|W| \leq |X| + |Y| \leq |X| + |Y| + |Z|$. Thus $|X'| \leq |Y'| + |Z'|$, and therefore ρ_2 is violated at the next iteration. □

Using this lemmas, we rewrite Algorithm 5 by unrolling some loops. More precisely, we obtained Algorithm 7 page 14 the following way:

- if ρ_1 is activated, then we merge X and Y and we are done.
- if ρ_2 is activated, then we merge Y and Z: the stack is now

$$\mathcal{X}' = (x_1, \ldots, x_{\ell-4}, W, X, \mathbf{merge}(Y, Z)).$$

  We unroll the loop once since Lemma 4 ensures that another rule is violated after the merge. We check whether ρ_1 is violated, if not we are sure that either ρ_2, ρ_3 or ρ_4 is not satisfied; in every of these cases X and $\mathbf{merge}(Y, Z)$ are merged. We just have to be careful to use $\mathcal{X}'$ for writing the nested conditions. For instance the nested condition for ρ_1 Line 13 is $|X'| < |Z'|$, for $\mathcal{X}' = (\ldots, X', Y', Z')$, which rewrites $|W| < |Y| + |Z|$.
- if ρ_3 is activated, then we merge Y and Z: the stack is now

$$\mathcal{X}' = (x_1, \ldots, x_{\ell-5}, V, W, X, \mathbf{merge}(Y, Z)).$$

By Lemma 5, we know that ρ_2 is violated on next iteration. Hence, we unroll once. The nested test for ρ_1 is done as previously. If ρ_1 is satisfied we know that ρ_2 is activated and then the stack is now

$$\mathcal{X}'' = (x_1, \ldots, x_{\ell-5}, V, W, \mathbf{merge}(X, Y, Z)).$$

We unroll once more (that is, three nested `if`), using the properties ensured when ρ_2 is activated, as before.

- if ρ_4 is activated, then we merge Y and Z and do not unroll the loop.

In this complicated version of TimSort, which is strictly equivalent to TimSort, we removed from C the costs of the merges that have been performed. What remains to prove, as we did for α-StackSort, is that Equation (2) holds after each update of C. This is done by induction. As C is always decreased just before ending an iteration of the main loop, we assume the property holds at the beginning of the `while` loop, and verify that it still holds when C is updated. There are seven cases, which we detail in the following.

For a given stack configuration $\mathcal{X} = (x_1, \ldots, x_\ell)$, let $f(\mathcal{X}) = \sum_{i \in [\ell]} 3i|x_i|$. By induction hypothesis, we assume that at the beginning of an iteration of the main loop, $C \geq f(\mathcal{X})$. We now check for the different cases, which is tedious but straightforward. $\mathbf{merge}(X, Y, Z)$ denote the result of merging X, Y and Z.

- **Line 10:** the stack goes from $\mathcal{X} = (x_1, \ldots, x_{\ell-3}, X, Y, Z)$ to

  $$\mathcal{X}' = (x_1, \ldots, x_{\ell-3}, \mathbf{merge}(X, Y), Z).$$

  Hence $f(\mathcal{X}) - f(\mathcal{X}') = 3|Y| + 3|Z|$. As rule ρ_1 is violated in this case, we have $|X| < |Z|$. Thus, the cost paid at Line 10 satisfies $|X| + |Y| - 1 < |Y| + |Z| < 3|Y| + 3|Z|$. Hence, the property still holds after Line 10.
- **Line 15:** $\mathcal{X} = (x_1, \ldots, x_{\ell-4}, W, X, Y, Z)$ becomes

  $$\mathcal{X}' = (x_1, \ldots, x_{\ell-4}, \mathbf{merge}(W, X), \mathbf{merge}(Y, Z)).$$

  Hence $f(\mathcal{X}) - f(\mathcal{X}') = 3|X| + 3|Y| + 6|Z|$. As $|W| < |Y| + |Z|$ in this case, the cost paid at Line 15 satisfies $|W| + |X| + |Y| + |Z| - 2 < |X| + 2|Y| + 2|Z| < 3|X| + 3|Y| + 6|Z|$. Hence, the property still holds after Line 15.
- **Line 18:** $\mathcal{X} = (x_1, \ldots, x_{\ell-4}, W, X, Y, Z)$ becomes

  $$\mathcal{X}' = (x_1, \ldots, x_{\ell-4}, W, \mathbf{merge}(X, Y, Z)).$$

  Hence $f(\mathcal{X}) - f(\mathcal{X}') = 3|Y| + 6|Z|$. As $|X| \leq |Y| + |Z|$ in this case, the cost paid at Line 18 satisfies $|X| + 2|Y| + 2|Z| - 2 < 3|Y| + 3|Z| < 3|Y| + 6|Z|$. Hence, the property still holds after Line 18.
- **Line 23:** This is exactly the same as for Line 15.
- **Line 28:** $\mathcal{X} = (x_1, \ldots, x_{\ell-5}, V, W, X, Y, Z)$ becomes

  $$\mathcal{X}' = (x_1, \ldots, x_{\ell-5}, \mathbf{merge}(V, W), \mathbf{merge}(X, Y, Z)).$$

  Hence $f(\mathcal{X}) - f(\mathcal{X}') = 3|W| + 3|X| + 6|Y| + 9|Z|$. As $|V| < |X| + |Y| + |Z|$ in this case, the cost paid at Line 28 satisfies $|V| + |W| + |X| + 2|Y| + 2|Z| - 3 < |W| + 2|X| + 3|Y| + 3|Z| < 3|W| + 3|X| + 6|Y| + 9|Z|$. Hence, the property still holds after Line 28.

– **Line 31:** $\mathcal{X} = (x_1, \ldots, x_{\ell-5}, V, W, X, Y, Z)$ becomes

$$\mathcal{X}' = (x_1, \ldots, x_{\ell-5}, V, \mathbf{merge}(W, X, Y, Z)).$$

Hence $f(\mathcal{X}) - f(\mathcal{X}') = 3|X| + 6|Y| + 9|Z|$. As $|W| \leq |X| + |Y|$ in this case, the cost paid at Line 31 satisfies $|W| + 2|X| + 3|Y| + 3|Z| - 3 < 3|X| + 4|Y| + 3|Z| < 3|X| + 6|Y| + 9|Z|$. Hence, the property still holds after Line 31.

– **Line 34:** $\mathcal{X} = (x_1, \ldots, x_{\ell-2}, Y, Z)$ becomes

$$\mathcal{X}' = (x_1, \ldots, x_{\ell-2}, \mathbf{merge}(Y, Z)).$$

Hence $f(\mathcal{X}) - f(\mathcal{X}') = 3|Z|$. As $|Y| \leq |Z|$ in this case, the cost paid at Line 34 satisfies $|Y| + |Z| - 1 < 2|Z| < 3|Z|$. Hence, the property still holds after Line 34.

We conclude as for α-StackSort: Equation (2) ensures that $C \geq 0$ when TimSort halts. Moreover, the sum of all increases of C is $\mathcal{O}(n \log n)$ and the number of comparisons is at most the sum of all decreases of C. Also, as for α-StackSort, the last stage of the algorithm where the remaining runs are merged, is $\mathcal{O}(n \log n)$. □

4.1 About TimSort and its variants

There are several reasons why TimSort has been adopted as a standard sorting algorithm in many different languages. An important difference between TimSort and other similar algorithms such as NaturalMergeSort or the algorithm given in [1] is the number of cache misses done during their execution. Indeed, in TimSort, runs are computed on the fly, and merges most often apply on the last few computed runs. Hopefully, they are still in the cache when they are needed. Analyzing cache misses is beyond the scope of this article, but we can notice that stack strategies of small degree like α-StackSort have the same kind of behavior, and should be cache-efficient too.

An interesting feature of α-StackSort is that the value of α can be chosen to improve its efficiency, provided we have some knowledge on the distribution of inputs. It is even possible to change the value of α dynamically, if the algorithm finds a better value in view of the first elements. The stack invariant can be violated if α is increased, but this does not affect the complexity of the algorithm.

Also observe that after designing a stack strategy, it is straightforward to take benefit from all the heuristics implemented in TimSort, as we just change the part where the rules are checked and the appropriate merges are performed.

5 Experiments and open question

We ran a few experiments to measure empirically the differences between TimSort and α-StackSort. Uniform random permutations of size $10,000$ were used for the first experiments, while we used sequences of exactly k runs in

n	k	TimSort	α-Stack
10,000	-	129,271.65	129,178.79
10,000	10	33,581.81	33,479.05
10,000	100	68251.65	67,154.79
100,000	100	678,449.70	669,692.66

Fig. 2. Average number of comparisons performed by merge sorting algorithms ($\alpha = 1.5$, Line 1: random permutations of size n, Lines 2-4: sequences of size n with k runs).

the second ones. The results, given in Fig. 2, indicates the cost **c** of the different merging strategies. We also checked on random permutations that the α-StackSort strategy performs as well as the implementation of TimSort in Java.[7]

It is quite natural to ask if the number of comparisons performed by TimSort is in $\mathcal{O}(n \log m)$ where m is the number of runs. This is the case for all the "non stack based" merge strategies mentionned here. It can be proved that it does not hold for α-StackSort, even though it should be easy to design a $\mathcal{O}(n \log m)$ version of this algorithm.

References

1. J. Barbay and G. Navarro. On compressing permutations and adaptive sorting. *Theor. Comput. Sci.*, 513:109–123, 2013.
2. S. de Gouw, J. Rot, F. S. de Boer, R. Bubel, and R. Hähnle. Openjdk's java.utils.collection.sort() is broken: The good, the bad and the worst case*. to appear in the *Proceedings of CAV 2015*. `http://envisage-project.eu/wp-content/uploads/2015/02/sorting.pdf`, 2015.
3. D. E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publish. Co., Redwood City, CA, USA, 1998.
4. T. Peters. Timsort description, accessed june 2015. `http://svn.python.org/projects/python/trunk/Objects/listsort.txt`.

[7] Benchmark using jmh: `http://openjdk.java.net/projects/code-tools/jmh/`

Algorithm 7: Main loop of unrolled-TimSort

```
1  X ← ∅
2  C ← 0
3  while R ≠ ∅ do
4      R ← pop(R)
5      Append R to X
6      C ← C + 3 |X| |R|
7      while X violates at least one rule of 𝔖 do
8          if |X| < |Z| then                                   /* ρ1 is activated */
9              Merge X and Y
10             C ← C − (|X| + |Y| − 1)
11         else if |X| ≤ |Y| + |Z| then                        /* ρ2 is activated */
12             Merge Y and Z
13             if |W| < |Y| + |Z| then                         /* ρ1 is activated */
14                 Merge W and X
15                 C ← C − (|W| + |X| + |Y| + |Z| − 2)
16             else                                            /* ρ2, ρ3 or ρ4 is activated */
17                 Merge X and merge(Y, Z)
18                 C ← C − (|X| + 2|Y| + 2|Z| − 2)
19         else if |W| ≤ |X| + |Y| then                        /* ρ3 is activated */
20             Merge Y and Z
21             if |W| < |Y| + |Z| then                         /* ρ1 is activated */
22                 Merge W and X
23                 C ← C − (|W| + |X| + |Y| + |Z| − 2)
24             else                                            /* ρ2 is activated */
25                 Merge X and merge(Y, Z)
26                 if |V| < |X| + |Y| + |Z| then               /* ρ1 is activated */
27                     Merge V and W
28                     C ← C − (|V| + |W| + |X| + 2|Y| + 2|Z| − 3)
29                 else                                        /* ρ2, ρ3 or ρ4 is activated */
30                     Merge W and merge(X, merge(Y, Z))
31                     C ← C − (|W| + 2|X| + 3|Y| + 3|Z| − 3)
32         else if |Y| ≤ |Z| then                              /* ρ4 is activated */
33             Merge Y and Z
34             C ← C − (|Y| + |Z| − 1)
```