Whamsort: an improved version of mergesort

Arnold Meijster[a,*], Wim H. Hesselink[a]

[a] *Bernoulli Institute for Mathematics, Computer Science and Artificial Intelligence, Nijenborgh 9, 9747 AG, Groningen, The Netherlands*

Abstract

The classical sorting algorithm *mergesort* is made more efficient by choosing a more appropriate merge procedure. Subsequently, a variation of the algorithm, called *whamsort*, is proposed, that in general is slightly more efficient than mergesort, but that is much more efficient when the input array is nearly sorted. Its worst-case time complexity ranges from linear for sorted input to order $N \log N$ for random input.

Keywords: sorting, mergesort, time complexity, correctness

1. Introduction

This short note has two messages. First, when the *merge* routine of the classical mergesort algorithm is replaced by a more efficient one, the time complexity of mergesort becomes linear for sorted input arrays. Secondly, it proposes *whamsort*, a variation of mergesort that, in general, performs as good or better than mergesort, but significantly better when applied to (nearly) sorted arrays.

To make this note self-contained, the mergesort algorithm is presented in section 2. In section 3 an improved version of the merge routine is presented which makes that the time complexity of mergesort becomes linear for sorted inputs. Whamsort is proposed in Section 4. In Section 5, it is proved that *whamsort* has a worst-case time complexity of the order of $\Theta(N \log N)$. Section 6 briefly discusses some related algorithms. Section 7 compares the performance on an ordinary PC. Section 8 draws conclusions.

2. Classical mergesort

This section presents the classical mergesort algorithm, which closely follows the exposition in [2, section 2.3.1]. In the following, the array to be sorted in ascending order is denoted x, and consists of N items of some type with an ordering operation. For simplicity, in this paper, x is assumed to be an integer array. We write $x[p, q)$ for the sequence of the elements $x[i]$ with the integer index i ranging in $p \leq i < q$.

The *mergesort* algorithm uses an auxiliary (scratch) array y of the same type and size as x, and a global call that hides the allocation of y and the recursion parameters:

```
procedure sort(N : int, var x[ ] : int);
    var y[N] : int; // allocate scratch memory
    mergesort(0, N, y, x);
end procedure;
```

The procedure call *mergesort*(p, q, y, x), sorts the subsequence $x[p, q)$ using y as scratch memory. If the subsequence contains more than one element (i.e. $q - p > 1$) then mergesort splits it into two parts, applies mergesort recursively on the parts, and finally merges the sorted subsequences using the routine *merge*.

```
procedure mergesort(p, q, y[ ] : int; var x[ ] : int);
    if q − p > 1 then
        var m : int;
        m := (p + q) div 2;
        mergesort(p, m, y, x);
        mergesort(m, q, y, x);
        merge(p, m, q, y, x);
    endif
end procedure;
```

For sorted subsequences $x[p, m)$ and $x[m, q)$, the call *merge*(p, m, q, y, x) (with $p < m < q$) merges

*Corresponding author

Email addresses: a.meijster@rug.nl (Arnold Meijster), w.h.hesselink@rug.nl (Wim H. Hesselink)

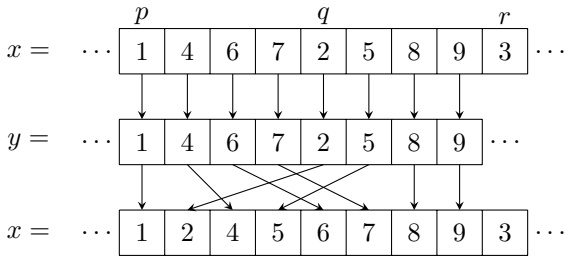

Figure 1: Data movement in the standard mergesort algorithm: *merge*(*p*, *q*, *r*, *y*, *x*).

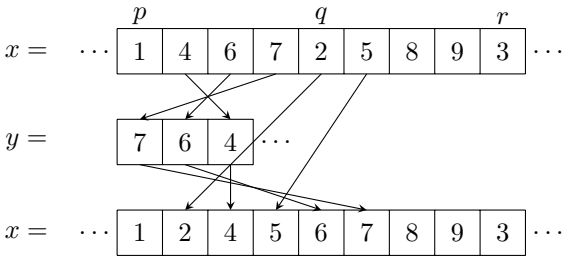

Figure 2: Data movement in the proposed merge algorithm: *merge*(*p*, *q*, *r*, *y*, *x*).

these sequences into a sorted sequence $x[p, q)$, using y as scratch memory.

```
procedure merge(p, q, r, y[ ] : int; var x[ ] : int);
    var i, j, k : int;
    for i := p to r − 1 do
        y[i] := x[i];
    endfor;
    i := p; j := q; k := p;
    while i < q and j < r do
        if y[i] < y[j] then
            x[k] := y[i]; i := i + 1;
        else
            x[k] := y[j]; j := j + 1;
        endif;
        k := k + 1;
    endwhile;
    while i < q do
        x[k] := y[i]; i := i + 1; k := k + 1;
    endwhile;
    while j < r do
        x[k] := y[j]; j := j + 1; k := k + 1;
    endwhile;
end procedure;
```

It is well-known that the worst-case time complexity of mergesort is $\Theta(N \log N)$, which is optimal for comparison sorting algorithms. However, mergesort always performs the same number of split and merge operations, regardless of the order of its input.

Moreover, the call *merge*(*p*, *q*, *r*, *y*, *x*) often performs redundant copying operations. It always moves a total of $2 \cdot (r-p)$ data items, because it first copies the entire subsequence $x[p, r)$ to the scratch memory y, and next moves these items from y back to x. However, some elements need no copying at all. For example, if $x[p] \leq x[q]$, then there is no need to move the element $x[p]$. Figure 1 shows a small example in which a total of 16 data movements take place. In section 3 an improved version of the *merge* routine is proposed that needs only 8 data movements for the same example.

3. An improved *merge* routine

In this section an improved version of the routine *merge* is proposed. It is optimized in such a way that, if *merge* need not move an element $x[i]$, the element is not involved in any assignment. The array y is used as a stack with stack pointer *sp*. The number of elements present in the stack is indicated by the value of *sp*.

```
procedure merge(p, q, r, y[ ] : int; var x[ ] : int);
    var sp, k : int;
    sp := 0; k := q;
    while p < k and x[q] < x[k − 1] do
        k := k − 1; y[sp] := x[k] ; sp := sp + 1
    endwhile;
    while sp > 0 do
        if q < r and x[q] < y[sp − 1] then
            x[k] := x[q] ; k := k + 1 ; q := q + 1;
        else
            sp := sp − 1 ; x[k] := y[sp] ; k := k + 1;
        endif
    endwhile
end procedure;
```

Note that short-circuit evaluation of guards is assumed (i.e. the second conjunct of a conjunction is evaluated only if the first conjunct evaluates to true).

The first loop of the routine *merge* determines the least index k (if existing) between p and q with $x[k] > x[q]$. Moreover, the loop copies the segment $x[k, q)$ with values that are greater than $x[q]$ to the stack $y[0, sp)$ in reverse order to make room for merging. Note that the loop terminates immediately if $x[q-1] \leq x[q]$ which corresponds with the situation that the entire segment $x[p, r)$ is sorted.

In that case the loop terminates with $sp = 0$, which means that the second loop will not be executed.

Figure 2 shows that, for the given example, only three items are copied to y, while the original *merge* routine copies eight items to y (see Fig. 1).

After termination of the first loop, it is clear that $p \leq k \leq k + sp = q \leq r$ holds. Moreover, for $p > k$ we have $x[k-1] \leq x[q]$. This means that the bag of elements to be merged is contained in the three subsequences $x[p,k)$, $x[q,r)$ and $y[0,sp)$. The elements of the subsequence $x[p,k)$ are already correctly placed. The second loop merges the subsequences $y[0,sp)$ and $x[q,r)$ into $x[k,r)$ in a way which is similar to the standard merge routine. It maintains $p \leq k \leq k + sp = q \leq r$ and $x[k-1] \leq y[sp-1]$ (for $k > p$). Once $sp = 0$, the loop terminates and the remaining items in $x[q,r)$ are not moved at all. In figure 2 that would be the situation in which $q = r - 2$, and the segment $x[q,r) = [8,9]$.

This implementation of *merge* has the time complexity $\Theta(r' - p' + 1)$, where $p \leq p' \leq r' < r$ and

$$r' = \mathrm{Max}\{j \mid x[j] < x[q-1] \ \vee \ j = q\}$$
$$p' = \mathrm{Min}\{i \mid x[q] < x[i] \ \vee \ i = q\}$$

For sorted input (i.e. $x[q-1] \leq x[q]$), this reduces to $p' = r' = q$ which means that the time complexity of the merge routine reduces to an $O(1)$ operation. As a result the time complexity of mergesort becomes linear in N for sorted input, while its worst case time complexity remains $\Theta(N \log N)$.

4. Whamsort

In this section *whamsort*, which is a variation on mergesort, is introduced. The name *whamsort* is derived from the initials of the authors.

While mergesort uses top down recursion, whamsort goes bottom up. The procedure call *whamsort*$(p,\ q,\ y\ , x)$, sorts the subsequence $x[p,q)$ using y as scratch memory. In an initiating loop, it first determines the longest sorted prefix of the input array, i.e. it searches for the largest r (with $p < r < q$) such that the subsequence $x[p,r)$ is sorted. The length of this prefix is $r - p$.

In a second loop, the value of r is incremented while maintaining the property that $x[p,r)$ is sorted. The loop stops when $r = q$. At the beginning of each iteration, the length of the sorted prefix is $l = r - p$. The body of the loop tries to double the size of the sorted prefix $x[p,r) = x[p,p+l)$ to $x[p,p+2\cdot l)$. The upperbound index of this extended prefix is assigned to the variable s, by setting $s = p + 2 \cdot l = p + 2 \cdot (r - p) = 2 \cdot r - p$. Clearly, if $s > q$, the value is reset to $s = q$. Next, the algorithm recursively sorts the subsequence $x[r,s)$ and merges the resulting sequence with the prefix $x[p,r)$ to obtain the sorted prefix $[p,s)$. Next, the assignment $r = s$ maintains the invariant of the loop and increases r. Note that, if the segment $x[p,q)$ contains less than two elements (i.e. $q - p \leq 1$), the procedure does nothing.

```
procedure whamsort(p, q, y[ ]:int; var x[ ]:int);
    var r, s : int;
    r := p + 1;
    while r < q and x[r − 1] ≤ x[r] do
        r := r + 1
    endwhile;
    while r < q do
        s := minimum(2 · r − p, q);
        whamsort(r, s, y, x);
        merge(p, r, s, y, x);
        r := s ;
    endwhile
end procedure;
```

5. Time complexity analysis of whamsort

In this section the time complexity of *whamsort* is determined. Let N be the number of items in the input array. Clearly, for a sorted input array the first loop of *whamsort* performs N iterations, and the second loop is not executed at all. Hence, the best case time complexity is $\Omega(N)$. However, its worst case time complexity is not immediately obvious. In the following analysis it is shown that the worst case time complexity of whamsort is $\Theta(N \log N)$. All logarithms are taken with base 2.

Clearly, the time complexity of *merge*(p,q,r,y,x) is linear in the size of the subsequence $x[p,r)$. Moreover, if m is the length of the longest sorted prefix of $x[p,q)$, then the time complexity of the first loop of *whamsort* is linear in m. Hence, we may assume that there is a constant α such that *merge*(p,q,r,y,x) performs at most $\alpha \cdot (r - p)$ basic operations, and that the first loop of *whamsort* performs at most $\alpha \cdot m$ operations.

Let $c(n)$ be the maximal number of operations that *whamsort* performs when called to sort $x[p,q)$ with $q = p + n$. We claim that $c(n) \leq 3 \cdot \alpha \cdot f(n)$ for all $n \geq 1$, where $f(n) = n \cdot \log(2 \cdot n)$. This is proved by induction. Clearly, the claim holds for

$n = 1$ because $f(1) = 1$. For the inductive step, it is assumed that $c(n) \leq 3 \cdot \alpha \cdot f(n)$ for all $n < N$. One has then to prove $c(N) \leq 3 \cdot \alpha \cdot f(N)$ for $N > 1$.

Consider a call of whamsort on an array of length N, say on $x[0, N)$. Assume that m is the length of the maximal sorted prefix of $x[0, N)$. Then $0 < m \leq N$. We may assume $m < N$, because the case $m = N$ corresponds to the best case in which the input is already sorted. Let k be the integer exponent such that $m \cdot 2^k < N \leq m \cdot 2^{k+1}$. The second loop of *whamsort* performs $k + 1$ iterations. In iteration i (with $0 \leq i < k$) it performs a recursive call on the segments $[m \cdot 2^i, m \cdot 2^{i+1})$. The last iteration performs a recursive call on the segment $[m \cdot 2^k, N)$. By the induction hypothesis, these recursive calls contribute to $c(N)$ at most

$$3 \cdot \alpha \cdot \left(f(N - m \cdot 2^k) + \sum_{i=0}^{k-1} f(m \cdot 2^i) \right)$$

operations. The calls of *merge* in the second loop contribute at most

$$\alpha \cdot \left(N + \sum_{i=0}^{k-1} m \cdot 2^{i+1} \right) = \alpha \cdot (N + 2 \cdot m \cdot (2^k - 1))$$

operations. Moreover, the first loop of *whamsort* contributes at most $\alpha \cdot m$ operations. Taken together, $\alpha \cdot (N + 2 \cdot m \cdot (2^k - 1)) + \alpha \cdot m \leq 3 \cdot \alpha \cdot N$, because $m \cdot 2^k < N$. It remains to prove that

$$N + f(N - m \cdot 2^k) + \sum_{i=0}^{k-1} f(m \cdot 2^i) \leq f(N),$$

because the lefthand side of this inequality multiplied by $3 \cdot \alpha$ is an upper bound of $c(N)$. The inequality can be rewritten as

$$N \leq \Delta$$

where $\Delta = f(N) - f(N - m \cdot 2^k) - \sum_{i=0}^{k-1} f(m \cdot 2^i)$.

For real numbers x, y, with $0 < x \leq y$ it holds

$$f(x + y) - f(x) - f(y) \geq 2 \cdot x$$

because the lefthand expression is increasing in y, and equals $2 \cdot x$ if $y = x$. This formula is applied with $x = N - m \cdot 2^k$ and $y = m \cdot 2^k$. Note that $x + y = N$. Moreover, $x \leq y$ because of $N \leq m \cdot 2^{k+1}$. The formula then gives that the value of Δ satisfies

$$\begin{aligned} \Delta &= f(x + y) - f(x) - \sum_{i=0}^{k-1} f(m \cdot 2^i) \\ &\geq 2 \cdot x + f(y) - \sum_{i=0}^{k-1} f(m \cdot 2^i) \\ &= 2 \cdot (N - m \cdot 2^k) + f(y) - \sum_{i=0}^{k-1} f(m \cdot 2^i) \end{aligned}$$

Hence, the proof requirement $N \leq \Delta$ reduces to

$$N - m \cdot 2^{k+1} + f(y) - \sum_{i=0}^{k-1} f(m \cdot 2^i) \geq 0$$

Since $m \cdot 2^k < N$, this inequality is surely satisfied if we replace N by $m \cdot 2^k$, so it suffices to prove

$$m \cdot 2^k - m \cdot 2^{k+1} + f(y) - \sum_{i=0}^{k-1} f(m \cdot 2^i) \geq 0$$

Recall that $y = m \cdot 2^k$ and that $f(n) = n \cdot \log(2 \cdot n)$, so all terms of this inequality have a common factor m. Hence, after some calculus it reduces further to

$$2^k \cdot \log(m \cdot 2^{k+1}) - \sum_{i=0}^{k-1} 2^i \cdot \log(m \cdot 2^{i+1}) \geq 2^k$$

which is equivalent to

$$2^k \cdot \log(m) + 2^k \cdot (k+1) - \sum_{i=0}^{k-1} 2^i \cdot \log(m \cdot 2^{i+1}) \geq 2^k$$

The third term can be rewritten as

$$\begin{aligned} & \sum_{i=0}^{k-1} 2^i \cdot \log(m \cdot 2^{i+1}) \\ = & \sum_{i=0}^{k-1} 2^i \cdot (\log(m) + \log(2^{i+1})) \\ = & \log(m) \cdot \sum_{i=0}^{k-1} 2^i + \sum_{i=0}^{k-1} 2^i \cdot (i+1) \\ = & (2^k - 1) \cdot \log(m) + (k-1) \cdot 2^k + 1 \end{aligned}$$

In the last step of this derivation the identities $\sum_{i=0}^{k-1} 2^i = 2^k - 1$ and $\sum_{i=0}^{k-1} (i+1) 2^i = (k-1) 2^k + 1$ were used. Substitution of this result in the above inequality yields after simplification that

$$\log(m) + 2^{k+1} - 1 \geq 2^k$$

This inequality clearly holds, which concludes the proof.

6. Related algorithms

In 1981, in EWD796a [3], Dijkstra proposed the algorithm *smoothsort* which is a variant on heapsort. It has a worst-case time complexity of order $\Theta(N \log N)$ and an almost linear behaviour if the input array is (almost) sorted. In the implementation of the algorithm, Dijkstra makes use of the so-called *Leonardo numbers* $L(n)$, which is a series of numbers that resemble the Fibonacci series. However, these numbers grow very fast, which might result in integer overflows for large inputs. A direct implementation of the algorithm using standard 32 bit integers will fail due to integer overflow for array with more than $L(32) = 7049155$ items. An implementation of smoothsort needs around 130 lines of C-code, while *mergesort* and *whamsort* are much simpler and both need less than 25 lines.

Currently, a popular sorting algorithm is *timsort* named after its inventor Tim Peters. It has the same best case and worst case time complexity as whamsort. Timsort is the sorting algorithm used in the standard library of the programming language Python. Peters did not publish the algorithm in an academic journal, but wrote an informal text on the internet explaining the details of the algorithm [4]. A more precise description and a formal worst case analysis was performed by Auger (in [1]). While *timsort* is a hybrid algorithm that combines mergesort with insertion sort and other optimizations, *whamsort* can be regarded as a straightened version where insertion sort and other optimizations have been removed.

7. Performance measurements

Timings were performed to compare the performance of *whamsort* with (the improved) mergesort and smoothsort. All algorithms were implemented in C, and were compiled using `gcc` (version 12.2, optimization level 3) on a PC with an Intel i7 processor (2.8 GHz clock) and 16GB memory.

Measuring performance differences between algorithms that have the same theoretical time complexity is a daunting task. To reduce fluctuations in the measurements, the best measurement out of 250 runs was taken for each algorithm. In table 1, the results are shown for an input array that was initialized with values that were generated by the random number generator. The column labeled N shows the number of items in the array, and the column labeled *time* shows the execution time (in milliseconds) of the standard mergesort algorithm (as published in [2]) for an array with N items. The numbers in the remaining columns are *speedups* relative to the standard mergesort algorithm. For example, a speedup of 1.14 means that an algorithm is 1.14 times faster than the standard mergesort algorithm. It is expected that speedup measurements are more stable with respect to compiler version and increased speed of CPUs than absolute time measurements. In the column *merge* the timings are shown for the version of mergesort that uses the improved merge routine from section 3. In the columns *wham* and *smooth* the results for whamsort and smoothsort are given. The program for *smoothsort* failed for $N > 10^6$ because of integer overflow. In view of the other results for smoothsort it was decided not to fix this issue.

N	*time*	*merge*	*wham*	*smooth*
10^5	9	1.15	1.19	0.57
10^6	104	1.14	1.15	0.51
10^7	1244	1.14	1.14	?
10^8	14241	1.14	1.14	?

Table 1: Sorting a random array of length N.

The measurements show that it pays off to replace the merge routine in mergesort by the improved merge routine from section 3. The performance increase is around 15%. Moreover, its performance is never worse than the orginal algorithm. As expected for random input, whamsort and mergesort score (almost) equally well. Smoothsort is clearly slower than all the other algorithms. This does not come as a surprise, because smoothsort is a variation on heapsort which has a much lower locality of memory references than mergesort. Especially on modern processors with caches, heapsort may result in frequent cache line misses, slowing down its performance.

In table 2, the number of items in the input array size is fixed at $N = 4 \times 10^6$. This size was chosen such that smoothsort still works without overflow using standard 32 bit integer arithmetic. Each row of the table is the mean of 100 measurements. For each measurement, an input array $x[\,]$ is constructed by initially setting $x[i] = i$ (for all i), followed by the application of k swaps of randomly chosen elements. Next, for that constructed array, time measurements were performed in the same style as for table 1 (best out of 100 runs for each algorithm).

swaps	*time*	*merge*	*wham*	*smooth*
0	107	6.47	41.73	4.78
1	119	5.55	19.36	4.59
10	123	3.86	6.67	4.40
100	128	2.71	3.50	4.14
1000	135	2.13	2.47	4.10
10000	142	1.76	1.93	3.95
100000	166	1.47	1.47	1.67
1000000	300	1.22	1.18	0.55
4000000	457	1.17	1.14	0.44

Table 2: Sorting a random permutation of length $N = 4 \times 10^6$

The time measurements were performed for k ranging from 0 to N. Hence, the input array is completely sorted for $k = 0$, while it is completely unordered for $k = N$. The first column in table 2 shows the number of applied swaps. The remaining columns show the same speedup information as in table 1.

The standard mergesort algorithm recursively splits the data, regardless of the order of the data items. However, the number of comparisons made in the merging phase is dependent on the structure of the input and is minimal for sorted input. This behaviour is clearly reflected in the column labeled *time* of table 2. For completely sorted input, the standard mergesort algorithm is almost four times faster for sorted input than for random input.

The improved mergesort algorithm that uses the merge routine from section 3 performs clearly better on input that is nearly sorted. However, the algorithm still recursively splits the data in the same way as in the standard mergesort algorithm.

As expected, the *whamsort* algorithm is clearly very effective on nearly sorted input. However, the increase in speed drops quite quickly if randomness is increased. Still, for an array in which around 1% of the data items are out of order, *whamsort* is (almost) twice as fast as the standard mergesort algorithm. Even though 1% seems to be small, there are several applications that would benefit from this performance increase. For example, in computer graphics applications in which polygons are rendered, it is crucial to determine which polygons should be drawn first, as polygons that are closer to the viewer should be rendered on top of those that are farther away. This is necessary to ensure correct occlusion when a three-dimensional scene is rendered to a two-dimensional display. In interactive graphics programs (like games), the position of the viewer changes minimally from one frame to the next frame, and as a consequence the list of polygons to be sorted for rendering also changes minimally. A change of less than 1 % is very common in this scenario, so *whamsort* is expected to reduce the time spent on sorting for this application.

In the last column of table 2, we see that *smoothsort* also performs better than *mergesort* if a small portion of the data is out of order. In fact, for the range $k \in [100..100000]$, *smoothsort* turns out to be the fastest algorithm. However, for larger k it performs even worse than standard mergesort. Overall, *whamsort* seems to be a better choice, because it is always faster than standard mergesort, it can cope with any array size, and scores well (albeit sometimes a bit less than smoothsort) on nearly sorted inputs.

8. Conclusions

For *mergesort* it pays to optimize the merge procedure. If one does this, mergesort takes only linear time on a sorted array.

The sorting algorithm *whamsort*, however, is even more efficient, especially on nearly sorted arrays. It is simple and it is stable. It could well be the algorithm of choice if the input array has some chance to be more or less sorted. For large random arrays it performs well, even slightly better than mergesort.

Dijkstra's *smoothsort* algorithm also performs well on nearly sorted arrays. However, for large unordered input arrays, its performance drops below the performance of standard *mergesort*. Moreover, the algorithm is much more complex than the other algorithms, and a correct implementation requires integer arithmetic with extended precision in order to avoid overflow.

References

[1] Nicolas Auger, Vincent Jugé, Cyril Nicaud, and Carine Pivoteau. On the Worst-Case Complexity of TimSort. In *26th Annual European Symposium on Algorithms (ESA 2018)*, volume 112, pages 4:1–4:13, 2018.

[2] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, 4th edition.* MIT Press, Cambridge, 2022.

[3] E.W. Dijkstra. Smoothsort, an alternative for sorting in situ. *Sci. Comput. Program.*, 1:223–233, 1982.

[4] T. Peters. *Timsort description, accessed june 2023.* svn.python.org/projects/python/trunk/Objects/listsort.txt, 2002.