decidedlyso / **merge-insertion-sort** Public

A Clojure implementation of the comparison-efficient Merge Insertion Sort / Ford Johnson Algorithm

11 stars 1 fork

Star Watch

Code Issues Pull requests Actions Projects Security Insights

master

rafd Adjust spacing in formulae ... on Sep 19, 2018 18

View code

Merge Insertion Sort / Ford-Johnson Algorithm

clojars [decidedlyso/merge-insertion-sort "1.0.2"]

This Clojure / ClojureScript library implements the Merge Insertion sorting algorithm (also known as the Ford-Johnson Algorithm).

Merge Insertion Sort is a comparison sort that minimizes the worst-case number of comparisons for small N (and has been proven optimal for N < 15, and likely optimal for N < 47).

This algorithm will not make your programs faster. Even though it has a better worst-case for small N than other algorithms, its relative complexity results in worse performance on actual computers than simpler algorithms (like quicksort, timsort and merge sort).

However, in situations where the cost-of-comparison greatly outweighs the cost-of-overhead of the algorithm, Merge Insertion Sort can be useful. For example, this implemention was motivated by needing an algorithm to determine the order in which to have (infinitely-slow) humans make pairwise comparisons.

It is also worth noting that just because Merge Insertion Sort has the best worst case (for small N), it doesn't mean it has the best average case (which, as far as I am aware, is an open problem).

Installation

Add the following dependency to your `project.clj` :

```
[decidedlyso/merge-insertion-sort "1.0.2"]
```

Usage

As with Clojure's built-in `sort` , you can optionally provide a comparator function that returns `-1/0/+1` or `true/false` :

```
(ns example
  (:require
    [merge-insertion-sort.core :as mi]))

=> (mi/sort [1 5 3 4 2 7 6 10 9 8])
(1 2 3 4 5 6 7 8 9 10)

=> (mi/sort > [1 5 3 4 2 7 6 10 9 8])
(10 9 8 7 6 5 4 3 2 1)

=> (let [comparisons (atom 0)]
     (mi/sort (fn [a b]
                (swap! comparisons inc)
                (compare a b))
              (shuffle [1 2 3 4 5 6 7 8 9 10]))
     @comparisons)
22
```

Background

Merge Insertion Sort was initially described by Ford and Johnson in 1959 (Ford 1959), but only took on its name when it was featured in the Art of Computer Programming (Knuth 1968). It is often referred to as the Ford-Johnson Algorithm.

The algorithm combines merging (like in merge-sort) and binary-search-insertion (like in insertion-sort), but, it is able to achieve better worst-case performance by better selecting which elements to compare, so as to maximize the efficiency of performing binary-search-insertion.

The key insight that underlies Merge Insertion Sort, is that it costs the same to perform binary-search-insertion on a list of `N = 2^K` as on a list of `N = 2^(K+1)-1` . For example, the worst-case for binary-search-insertion for `N = 8` is `floor(log2(N)) = 3` , and it is the same for `N = 9 to 15` . When given a choice between elements to compare, the algorithm prefers pairs that would require inserting an element into a list of `N = 2^K-1` (ie. a list with length one less than a power of 2), because it can insert that element for one less comparison than if that insertion were to be made one insertion later.

It turns out that the order of such comparisons can be determined by an integer progression called the Jacobsthal numbers, at least when optimizing for the worst-case.

The Merge Insertion Sort algorithm is as follows:

1. Given an unsorted list, group the list into pairs. If the list is odd, the last element is unpaired.

2. Each pair is sorted (using a single comparison each) into what we will call [a b] pairs.

3. The pairs are sorted recursively based on the `a` of each, and we call the pairs [a1 b1], [a2 b2] etc. If the list was odd, the unpaired element is considered the last `b` .

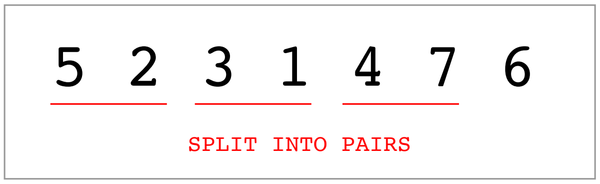
4. We call the chain of `a` s the "main-chain".

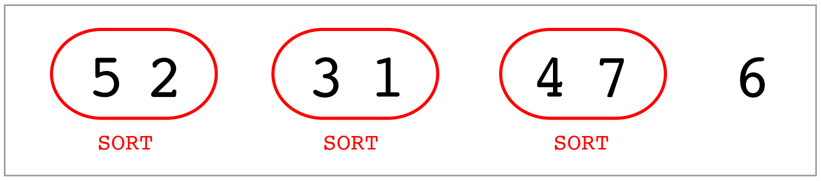
   At this point, we could take any of the `b` s and use binary-search-insertion to insert that `b` into the main-chain (which starts of as just the `a` s). When inserting, we only need to consider the values "left" of the `b` in question (for example, when inserting `b4` we only need to consider the chain up to and including `a3`).

> We could insert the `b` s in order (`b1` , `b2` ...), but the "key insight" from above suggest otherwise. Different `b` s have different worst-case costs to insert into the main-chain (worst case cost for binary-search-insertion is floor(log2(N) where N is the length of the relevant part of the main-chain). We can minimize the cost by following an order based on the Jacobsthal Numbers: *1* *3* 2 *5* 4 *11* 10 9 8 7 6 *21* 20 19 18... (ignoring values which are greater than the `b` s we have).
>
> And so, we insert the `b` s, one at a time, into the main-chain following the above progression, eventually resulting in a sorted list.

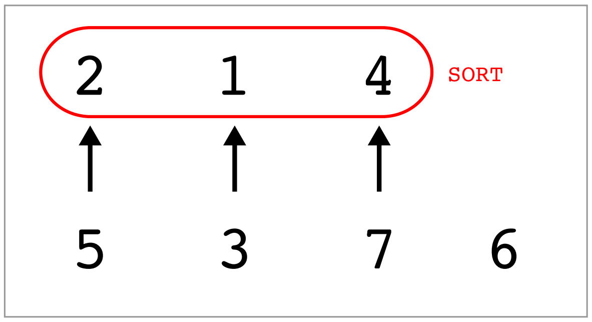
Below is an example that walks through the above steps:

worst case
# of comparisons

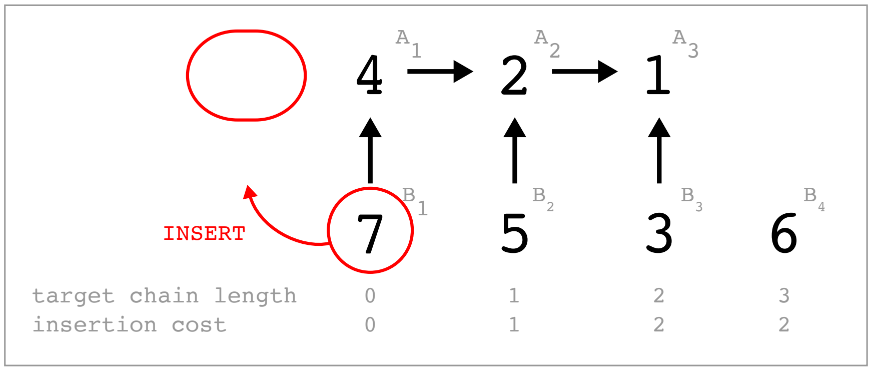




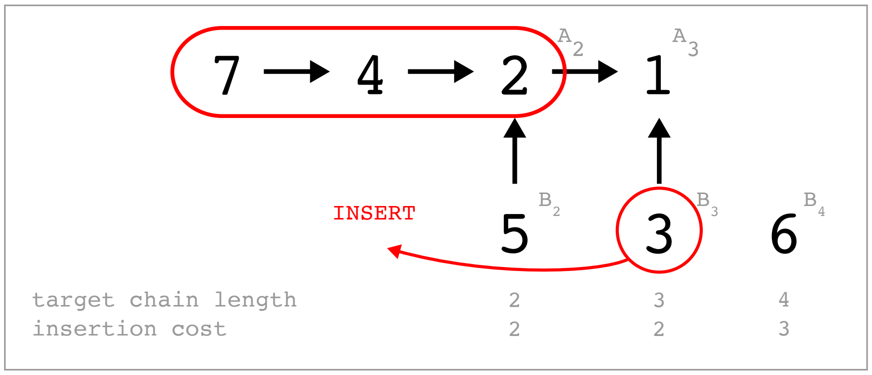
3



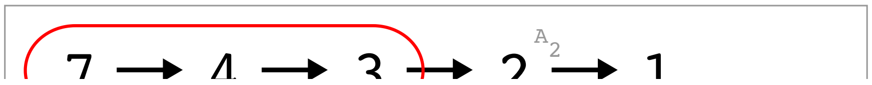
3



0



2



If the above explanation was confusing, peruse the code, or try: (Ford 1959), (Knuth 1968), (Ayala-Rincon 2007), or (JaakkoK 2009). A C++ implentation also exists (Morwenn Github 2016) with an explanation by the author (Morwenn StackExchange 2016).

Performance

From information theory, the lower bound for the minimum comparisons needed to sort a list is: `⌈log2(N!)⌉`

As derived in (Knuth 1968), the worst case comparisons for Merge Insertion Sort is: `sum(k=1..N) of ⌈log2(3k/4)⌉`.

For comparison, worst case for binary-search-insertion is `sum(k=1..N) of ⌈log2(k)⌉`.

In a table, this gives:

n	1	2	3	4	5	6	7	8	9	10	11	12
worst case lower bound	0	1	3	5	7	10	13	16	19	22	26	29
merge-insertion worst case	0	1	3	5	7	10	13	16	19	22	26	30
merge-insertion optimal by information theory	*	*	*	*	*	*	*	*	*	*	*	
merge-insertion optimal by exhaustive search												*
binary-search-insertion worst case	0	1	3	5	8	11	14	17	21	25	29	33

n	1	2	3	4	5	6	7	8	9	10	11	12
`⌈N∗log2(N)⌉`	0	2	5	8	12	16	20	24	29	34	39	44

Merge Insertion Sort's worst-case is equal to the information theoretic minimum, and thus optimal, for N = 1 to 11, 20 and 21.

Via exhaustive search on computers, Merge Insertion Sort was proven optimal for N = 12 (Wells 1965), 13 (Kasai 1994), 14, 15 and 22 (Peczarski 2004).

Whether it is optimal for N = 16 to 19 is still an open question (Peczarski 2012), but no better algorithm has been found for N < 47 and it has been proven that it would need to rely on a completely different strategy (Peczarski 2007). There are modified algorithms that perform similarly to Merge Insertion Sort (Ayala-Rincon 2007), and ones that are better at N > 47 (Manacher 1979, Bui 1985, Manacher 1989).

The minimum number of comparisons is also tracked in a list in the Online Encyclopedia of Integer Sequences (Sloane 2017).

References

- **(Ayala-Rincon 2007)** M. Ayala-Rincon, B. T. de Abreu, J. de Sequira, A variant of the Ford-Johnson algorithm that is more space efficient. Inf. Proc. Letters, 102, 5 (2007) 201–207. https://www.researchgate.net/publication/222571621_A_variant_of_the_Ford-Johnson_algorithm_that_is_more_space_efficient
- **(Bui 1985)** T. D. Bui, M. Thanh, Significant improvements to the Ford-Johnson algorithm for sorting, BIT, 25 (1985) 70–75. https://link.springer.com/article/10.1007/BF01934989
- **(Ford 1959)** L. Ford, S. Johnson, A tournament problem, American Mathematical Monthly 66 (1959) 387–389. https://www.jstor.org/stable/2308750
- **(JaakkoK 2009)** https://stackoverflow.com/questions/1935194/sorting-an-array-with-minimal-number-of-comparisons/1935353#1935353
- **(Kasai 1994)** T. Kasai, S. Sawato, S. Iwata, Thirty Four Comparisons Are Required to Sort 13 Items, LNCS, 792, (1994) 260–269.

- **(Knuth 1968)** D. Knuth, The Art of Computer Programming, Volume 3, Section 5.3.1 (1968). Relevant extract: https://www2.warwick.ac.uk/fac/sci/dcs/teaching/material/cs341/FJ.pdf
- **(Manacher 1979)** G. K. Manacher, The Ford-Johnson sorting algorithm is not optimal. J. ACM, 26, 3 (1979) 441–456.
- **(Manacher 1989)** G. K. Manacher, T. D. Bui, T. Mai, Optimal combinations of sorting and merging, J. ACM 36 (1989) 290–334.
- **(Morwenn Github 2016)** https://github.com/Morwenn/cpp-sort/blob/master/include/cpp-sort/detail/merge_insertion_sort.h
- **(Morwenn StackExchange 2016)** https://codereview.stackexchange.com/questions/116367/ford-johnson-merge-insertion-sort
- **(Peczarski 2004)** M. Peczarski, New Results in Minimum-Comparison Sorting, Algorithmica, 40, 2 (2004) 133-145.
- **(Peczarski 2007)** M. Peczarski, The Ford-Johnson algorithm still unbeaten for less than 47 elements, Inf. Proc. Letters, 101, 3 (2007) 126–128.
- **(Peczarski 2012)** M. Peczarski, Towards optimal sorting of 16 elements. Acta Univ. Sapientiae, Inform. 4, 2 (2012) 215–224. https://arxiv.org/pdf/1108.0866.pdf
- **(Sloane 2017)** N. J. A. Sloane, Sorting numbers: minimal number of comparisons needed to sort n elements, The Online Encyclopedia of Integer Sequences. https://oeis.org/A036604
- **(Wells 1965)** M. Wells, Applications of a Language for Computing in Combinatorics, Information Processing, 65, (1965) 497–498.

Releases

3 tags

Packages

No packages published

Languages

- ● **Clojure** 100.0%