ford-johnson-merge-insertion-sort

An implementation of the Ford-Johnson Merge-Insertion Sort Algorithm in Python

Background:

While there are several well-known sorting algorithms in wide-use today, they often gauge their success on the speed of the sort and their ease in implementation rather than reducing the number of comparisons needed during any given sort.

The Ford-Johnson Merge-Insertion sorting algorithm focuses on solely this aspect, providing an algorithm in which the least amount of comparisons is needed. Developed in 1959 by Lester Ford and Selmer Johnson, this algorithm has set the standard for low comparison sorting and continues to act as the bedrock for all research in that area of study.

We will provide a historical background on the Ford-Johnson Merge-Insertion Sort, an implementation of the algorithm in Python, background on why it was developed and what it is used for, provide an overview of the given implementation, a look as to how efficiency is measured for this algorithm, a review of what metrics and tests were used when measuring the efficiency in this implementation and what was learned as a result.

The implementation can easily be imported into any project and used. Tests are included in 'test.py'. All bugs and pull requests welcomed.

History

Lester R. Ford Jr and Selmer M. Johnson first developed Ford-Johnson sorting algorithm in May 1959, while employed at the RAND corporation with the publication of the article “A Tournament Problem” in the American Mathematical Monthly (Ford and Johnson). The article was focused on addressing the fundamental question of how to perform a sort with the least amount of comparisons -- or, as Ford and Johnson put it in a sportsmanship-like context, “What is the smallest number of matches which will always suffice to rank all n players?” (Ford and Johnson).

The designed algorithm was first titled ‘Ford-Johnson tournament sort’ and subsequently re-named (and popularized) by Donald E. Knuth with the more generalized name ‘Merge-Insertion sort’ to account for the possible concurrent independent discovery by S. Tybula and P. Czen in 1959 (Knuth, 184-187). After its publication by Ford and Johnson, it was successful in setting the standard for the lowest number of comparisons needed to perform a sort for over 20 years (Manacher).

However, while successful in achieving the “informative-theoretic” bound (Manacher), the algorithm has served little practical use as it requires complex data structures and is slower to execute than more common and simpler to implement sorts (Morwenn).

Regardless of this, Ford-Johnson Merge-Insertion Sort (FJMI) has retained its status as the foundational standard for sorting algorithms which target the fewest number of comparisons, and all subsequent research and improvements on sorts with this bound targeted have been built upon FJMI sort and this is likely to continue (Mahmoud, 286).

Implementation:

The implementation of FJMI sort accompanying this paper was written in Python and follows the following steps:

- Determine if the array is even or odd numbered in length. If odd, remove the last number, designate it as a ‘straggler’ and insert it later into the sorted array.

- Arbitrarily divide the sequence to sort into pairs of two values.
- Sort the pairs bitwise, so the order is always [less, greater].
- Sort the sequence recursively by the value of it's largest pair.
- Create a new sequence 'S', by pulling out the [highest] value of each pair and inserting it into 'S'.
- The remaining values form a temporary 'pend' array.
- Based on the length of 'pend', build the optimal insertion sequence using relevant Jacobsthal numbers.
- Loop through the elements in 'pend', and using the insertion sequence built in the previous step, use binary search to insert each 'pend' element into 'S'.
- If a 'straggler' was found, do a leftover loop and insertion to complete the list.

Step 1: 'Straggler' Catching

In order to determine if the given array is even or odd, the operation requires a simple modulo calculation. This is performed within the body of 'merge_insertion_sort' against the incoming array and is saved as a boolean value.

If the operation proves it's an odd-number length array, we 'pop' the last value from the incoming array arbitrarily and save it into a local value. If we find it's even, we set the value to 'False'. We'll use this value later during 'create_s' as an optional parameter where it will be placed in the ultimately sorted array.

Step 2: Creating Pairs

In order to structure the data to follow FJMI, the values in the array need to be arbitrarily sorted into pairs. We do this by using 'create_pairs', which loops through the value, and using temporary array stores, creates 'pairs' with two element arrays which are inserted into a larger array that is returned.

Step 3: Sorting each pair

Once the array we need to sort is paired, we use 'sort_each_pair' and pass in the paired array. This function loops through the pairs, and does a simple bitwise comparison placing the smaller number on the left and the larger on the right, and returns the array to the user.

Step 4: Sort the Pair Sequence by its greater value

The next step in FJMI is to recursively sort all the pairs by their largest element. In this implementation, we use ‘sort_by_larger_value’, which receives the array (which has already been split into pairs, and each pair sorted itself) and uses a modified recursive insertion sort (using functions ‘insertion_sort_pairs’ and ‘insert’) that perform this action. This leaves the array sorted by the greater value in each pair.

Steps 5-9: Creating sorted ‘S’ sequence

The following steps contain the most complex aspects of FJMI. In this implementation, these steps are housed in ‘create_s’, which takes the array output by ‘sort_by_larger_value’ and the previous steps, as well as the ‘straggler’ value mentioned in step one if applicable, and returns the completed sorted sequence.

The function first creates the ‘main’ array (we call it ‘S’) which we will eventually return, as well as ‘pend’, which will be a temporary hold used during FJMI. After creating both empty arrays,, it loops through the incoming paired array, placing the larger values (which have been sorted) into ‘S’, and the remaining smaller elements into ‘pend’.

Immediately, since we know the first element in ‘pend’ (we call it: ‘p1’) is smaller than the first element in the ‘S’ array (‘s1’), we insert it in the ‘0’ index of S.

After this step, we have to create the sequence in which we can optimally insert the remaining ‘pend’ elements in a sequence that leverages the fact that in a worst-case scenario, binary search performs best when the sub-sequence searched is one-less than the power of two. As Knuth discovered, the pattern here corresponds with Jacobsthal numbers -- a sequence we’ll build with ‘build_jacob_insertion_sequence’.

This function takes the ‘pend’ array, and using it’s length, leverages a recursive Jacobsthal generation function ‘jacobsthal’, builds and returns an array of the relevant Jacobsthal numbers which will be used to build the full insertion sequence.

The sequence commences by looping through the ‘pend’ array, dynamically determining the full sequence (it must use the Jacobsthal numbers, filled in with the remaining values depending on context), subsequently determining the exact insertion point and finally inserting each value into S, taking advantage of the above key efficiency in FJMI. This sequence begins to look like [1, 3, 2, 5, 4, 11...], continuing on with depending on the length on the ‘pend’ array.

Finally, if a ‘straggler’ was passed into this function, we perform a binary search against the entire ‘S’ sequence and insert it into S as well, before returning the final sorted array.

Efficiency

As previously mentioned, the key goal of FJMI is to limit the number of comparisons that are needed to perform a sort (Ford and Johnson), mainly by taking advantage of the efficiency of inserting with a sequence that limits the binary insertion search area where the length needing to be searched is one less than the power of two (Decidedlyso). It is

README.md

It is also known that, beside its complexity in implementation, one of the major reasons why FJMI is not widely adopted is it's speed in computational performance (Morwenn). Even with the understanding that FJMI's speed in performance is not it's target measure of efficiency, there is still surprisingly little publicly available data on this topic.

As a result, as a means to gauge the efficiency of this implementation, an approximate number of comparisons per sort is gathered during runtime. In addition, to address the shortfall of data on computational performance, a basic time duration test is also run during sort and is included in this analysis.

Performance Test

To get the key metrics outlined above, this implementation includes an estimated 'comparision_counter' that is printed to the user. This is driven by a boolean parameter in 'create_s' named 'print_comparision_esimatation', as this is the main function which encapsulates the comparisons for the final sort.

Since the number of comparisons needed in a binary search of three items is log2(3), or approximately 2, the counter increments by 2 with each sequenced insertion, given the sequence inherently limits the subsequence needed for insertion to need no more than 3 items.This estimation is also included when a straggler is inserted.

To test this across scale, a 'create_array' function is included that takes a desired 'length' and outputs an array of random, non-repeated integers to be sorted. Once generated, we start a timer, run the sort (which prints the comparison count by default) and when complete, do a simple time duration calculation to determine the time it took for the sort to complete.

Performance Test Metrics

Number of Items	Number of Approximate Comparisons Made	Duration of Sort Execution (Seconds)
10	8	0.00007629394
20	18	0.0001442432403564453
100	90	0.0004258155822753906
1000	844	0.01973128318786621
2500	1958	0.09786701202392578
5000	N/A	N/A

Analysis of Test Results

The implementation of the FJMI algorithm performed well when these tests were applied, especially with lower numbered sets, as shown in the results above. While the approximate comparisons were not as close to the minimums referenced by Knuth, they are within range. For example, when N is 12 this implementation approximates 10 comparisons while the minimum should be 9. The speed of performance was surprisingly fast as well; though likely not as fast as other sorts, it did not perform with significant lag as one would expect.

When the set size increased, the number of comparisons increased significantly and performance slowed; both are expected and are known shortfalls of the FJMI sort algorithm, which as previously mentioned performs best with smaller sets (Knuth).

A surprising find during these tests was what happened when the set was significantly long, and where the failure occurred. When a large set was given (on average, anything over 2525 entries), the implementation began to fail by means of reaching the maximum recursion depth, while sorting the pairs. While not an algorithmic limit, it does show that the complexity in implementation does put a greater load on the computer, which can cause errors when sorting large data sets where other sorting algorithm implementations would not.

Conclusion:

As shown in both this analysis, the corresponding implementation of Ford-Johnson Merge-Insertion sort (FJMI), and the metrics in efficiency gathered, the algorithm performs optimally in reducing the number of needed comparisons in sorting particularly in shorter data sets. While it performs less efficiently, both in terms of needed comparisons and the scaling it's complex implementation, FJMI continues to provide insight as to optimal sorting algorithms and theoretical research in this area.

Although it's implementation will likely continue to not be practical when compared to other sorting algorithms, FJMI will likely continue to standard as the basis of all low-comparison sorting, providing valuable insights and spur continued improvements and research for sort optimization in the future.

Citations:

Decidedlyso. "Merge-Insertion-Sort". Github. https://github.com/decidedlyso/merge-insertion-sort

Ford, Lester R., and Selmer M. Johnson. "A Tournament Problem." The American Mathematical Monthly, vol. 66, no. 5, 1959, pp. 387–389. JSTOR, www.jstor.org/stable/2308750. Accessed 13 June 2020.

Knuth, Donald E. The Art of Computer Programming: Volume 3: Sorting and Searching. Addison-Wesley Professional; 2 edition (May 4, 1998).

Mahmoud, Hosam M. Sorting: A Distribution Theory. John Wiley & Sons. (October 14, 2011).

Manacher, Glenn K. 1979. "The Ford-Johnson Sorting Algorithm Is Not Optimal". J. ACM 26, 3 (july 1979), 441–456. DOI:https://doi.org/10.1145/322139.322145

Morwenn. "Ford-Johnson Merge-Insertion Sort". Code Review Stack Exchange. 10 Jan. 2016, https://codereview.stackexchange.com/questions/116367/ford-johnson-merge-insertion-sort

Releases

No releases published

Packages

No packages published

Languages

● **Python** 100.0%