A Variant of the Ford-Johnson Algorithm that is more Space Efficient

Mauricio Ayala-Rincón[1]*, Bruno T. de Abreu[2], and José de Siqueira[2]

[1] Departamento de Matemática, Universidade de Brasília, Brasília D.F., Brasil
ayala@mat.unb.br

[2] Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Belo Horizonte, Brasil {jose,brunotx}@dcc.ufmg.br

Abstract. A variant of the Ford-Johnson or *merge insertion* sorting algorithm that we called **four** Ford-Johnson ($_4$FJ, for short) is presented and proved to execute exactly the same number of comparisons than the Ford-Johnson algorithm. The main advantage of our algorithm is that, instead of recursively working over lists of size the half of the input, as the Ford-Johnson algorithm does, $_4$FJ recursively works over lists of size the quarter of the input. This allows for implementations of data structures for coordinating the recursive calls of size only 33% of the ones needed for the Ford-Johnson algorithm.

Keywords: analysis of algorithms, sorting algorithms, optimal sorting.

1 Introduction

Searching for optimal sorting algorithms is a fascinating field of research in computer science which in short will complete fifty years [3, 2]. One of the most relevant and nice pieces of work in this direction is the *merge insertion* algorithm discovered by Lester Ford and Selmer Johnson, that we will call the Ford-Johnson or FJ algorithm, for short [3]. This algorithm gives rise to a lot of work on developing optimal sorting algorithms as close as possible to the information-theoretic lower bound of $\lceil log_2 n! \rceil$ comparisons for sets of n keys. The question *What is the best possible way to sort?* is pointed out in the famous Donald Knuth third volume on *Sorting and Searching* of *The Art of Computer Programming* [4] (page 180 of the second edition) focused mainly in optimizing the number of comparisons. There the Ford-Johnson algorithm was described and the number of comparisons the algorithm makes were compared with the theoretical lower bound. After this work, the running time of the FJ algorithm has been proved not optimal: in [6] it is proved that the FJ algorithm can be beaten for infinitely many values of n starting with $n = 189$ and in [9] starting with $n = 47$. The method presented in [6] has been improved in [7] obtaining sorting of lists of 52 keys with 230 comparisons: one less than with the FJ algorithm. But an optimum algorithm even for sets of small size (such as 14, 15, or 16 elements) remains unknown [5]. Progress in this direction has been obtained recently: although the theoretical lower bound is 33, in [8] it is proved that sorting 13 keys requires 34 comparisons using a refinement of Wells algorithm [10]. This is the number of comparisons needed as well by the FJ (and our variant) algorithm.

* Author partially supported by the Brazilian CNPq Council.

The FJ algorithm has not been of practical use because of the inherent complexity of the data structures needed for its implementation. In particular, this algoritm requires an administrative memory for the recursive bookkeeping which consists of the memory needed in each recursive environment for providing a sublist of keys (of size the half of the list of the environment) as argument of one recursive call and the memory necessary for maintaining the changes of positions of these keys after the recursive call returns the ordered list. The former corresponds to the necessary memory used by data structures such as lists of words of the same size used by the keys being sorted and the latter corresponds to the necessary memory used by structures such as lists of pointers. The space used by these data structures is defined as the administrative or bookkeeping memory. The variant of the FJ algorithm that we propose in this paper is proved to execute exactly the same number of comparisons between keys than the Ford-Johnson algorithm, but the required administrative memory is smaller, since it executes at most the half of the recursive calls (over lists of size the quarter of the lists of the recursive environments) that FJ executes and these recursive calls are invoked over a total of keys and pointers corresponding only to 33% of the data processed by the recursive calls of the FJ algorithm.

After briefly describing the FJ algorithm in the section 2, our variant, the ${}_4$FJ algorithm, is introduced and its running time is analyzed counting the needed number of comparisons in section 3. Then the FJ and ${}_4$FJ running time and administrative memory are compared in section 4 before concluding and discuss future work.

2 Merge Insertion or the Ford-Johnson algorithm (FJ)

A brief description of the *merge insertion* or FJ sorting algorithm is given, which is based in the elegant presentation of D. Knuth [4] (page 184 of the second edition).

The key idea of the FJ algorithm is to explore binary insertion maximally: it is better to binary insert a key in a sorted list of size $2^{k+1} - 1$ than in a list of size 2^k because in both cases $k + 1$ comparisons are needed. Notice that this is not explored by well-known efficient sorting algorithms such as *binary insertion sort*: when ordering, for instance, a list of five different keys the third placed key is inserted in a sorted list of size two (with two comparisons in the worst-case) and the last placed key in a list of size four (with three comparisons in the worst-case). By applying binary insertion sort, a list of five different keys is sorted with a total of 8 comparisons in the worst-case: $\sum_{i=1}^{5} \lceil log_2(i) \rceil$, which is bad. To explore binary insertion maximally over five keys (see the Figure 1), say a, b, c, d, e, we firstly insert isolated keys over unitary lists obtaining an ordering structure of the form $a < b, c < d, e$. From that point, notice that inserting the remaining isolated key e into one of the two sorted lists of size 2 is not convenient, because is better to binary insert a key in a list of size 3 (since we will need two comparisons in the worst-case, for both). Then we compare the two greater keys of the two ordered lists of size two, obtaining the quadruplet structure illustrated in the third step of the Figure 1. Observe that this step resumes to binary insertion in a unitary list. Then the remaining isolated key, e, can be inserted in a list of size three, giving as result either the situation illustrated in the fourth step of the figure or a simpler one where the key e results greater than the greatest key of the quadruplet. Finally, in the fifth step, the pending key is binary inserted in a list of size two or three. Observe that in the later two steps binary insertion is explored maximally too.

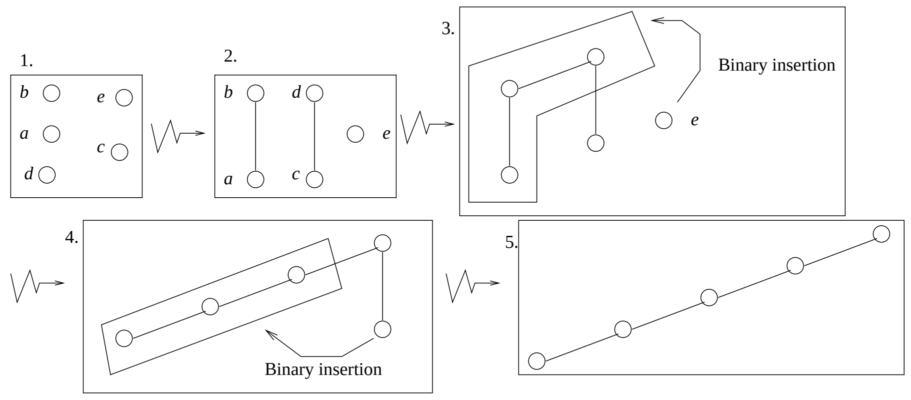

Fig. 1. Ordering five keys exploring binary insertion maximally.

The FJ algorithm sorts a set of keys of size n by exploring binary insertion maximally according to the following three steps.

1. The given input set of keys is divided into $\lfloor n/2 \rfloor$ subsets of two keys that are ordered as pairs, whose greater and smaller keys are denoted as *max* and *pend* elements, respectively. The remaining key, in the case n is odd, is considered a *pend* element.
2. Recursively, order the set of $\lfloor n/2 \rfloor$ *max* elements obtaining a structure as the one presented in the Figure 5.
3. In this step the "hanging" $\lceil n/2 \rceil$ *pend* elements are inserted in the upper-line ordered list consisting of the *max* elements, that is called the *main chain*. The *pend* elements are inserted in the main chain by exploring binary insertion maximally, as follows:
   - the left-most *pend* element is yet in the correct order.
   - The third *pend* element is selected as pivot and inserted in the list of size three containing the two left-most *max* elements and the left-most *pend* elements. Then the second left-most *pend* element is inserted in a list of size less than or equal to three, which contains the left-most *max* and *pend* elements, and, probably, the second *pend* element. Now, the main chain consists of all the *max* and the third left-most *pend* elements.
   - The fifth *pend* element is selected as pivot and can now be inserted into its proper place in the first seven elements of the main chain and after this, the fourth *pend* element is inserted into its proper place in the first segment of the main chain to the left of its associated *max* element. Notice that this list is limited to seven elements too. Now, the main chain consists of all the *max* and the fifth left-most *pend* elements.
   - The eleventh *pend* element is selected as pivot and can now be inserted into its proper place in the segment of the main chain consisting of the ordered ten left-most *max* and the five left-most *pend* elements, which is a sorted list of size fifteen. Then the ninth, the eight, the seventh and the

sixth left-most *pend* elements are inserted in lists limited to size fifteen. Now, the main chain consists of all the *max* and the eleventh left-most *pend* elements.
- The remaining *pend* elements are inserted following this order, by selecting adequate "pivots", which maximize the application of binary insertion.

The worst-case number of comparisons executed by the FJ algorithm for an input of size n, $F(n)$, as denoted in [4] is given by the equation

$$F(n) = \lfloor \frac{n}{2} \rfloor + F(\lfloor \frac{n}{2} \rfloor) + G(\lceil \frac{n}{2} \rceil) \tag{1}$$

where, $G(\lceil n/2 \rceil)$ denotes the number of comparisons needed for inserting $\lceil n/2 \rceil$ keys into a main chain of $\lfloor n/2 \rfloor$ elements. $G(m)$ is defined from the sequence that defines the "pivots" of the order of the binary insertion of "hanging" keys into a main chain: $t_1 = 1, t_2 = 3, t_3 = 5, t_4 = 11, \ldots$. More specifically,

$$t_k = \begin{cases} 1 & \text{if } k = 0 \text{ or } k = 1, \\ 2^k - t_{k-1} & \text{if } k > 1. \end{cases} \tag{2}$$

For $k > 1$, t_k can be proved equal to $(2^{k+1} + (-1)^k)/3$ [4].

Equation 1 can be simplified obtaining

$$F(n) = \sum_{k=1}^{n} \lceil log_2(\frac{3}{4}k) \rceil \tag{3}$$

3 Variant of the Ford-Johnson algorithm: $_4$FJ

The main difference between the FJ and $_4$FJ algorithms is that the FJ algorithm works recursively over a list of size the half and the $_4$FJ over a list of size the quarter of the initial list. Binary insertion of keys over ordered lists follows the same strategy in both methods.

We will describe one step of the recursive $_4$FJ and then we will illustrate how it works over a list of specific size.

In each recursive call $_4$FJ divides the input list into disjoint subsets of **four** keys and builds basic ordered structures of the form presented in the Figure 2 called *quadruplets*. Each of these quadruplets is built with only three comparisons. According to their position in the quadruplet the elements are called maximum *max*, pending *pend*, least than pending *ltp* and least than maximum *ltm*.

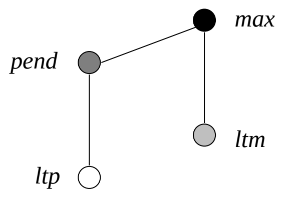

Fig. 2. A *quadruplet*, the basic ordered structure of the $_4$FJ: $max \geq ltm$ and $max \geq pend \geq ltp$.

Our algorithm consists of the following four steps

1. The given input set of keys is divided into $\lfloor n/4 \rfloor$ subsets of four keys that are ordered as quadruplets. In the case $n \bmod 4 \geq 2$, two of the remaining keys are ordered being the greatest key considered a pending key and the other its *ltp* element and the third one (when $n \bmod 4 = 3$) is considered a *ltm* element. When $n \bmod 4 = 1$, the remaining key is considered a *ltm* element.
2. Recursively order the set of size $\lfloor n/4 \rfloor$ consisting of *max* keys of the quadruplets (see Figures 3 and 4).
3. In this step, we will consider only the structure consisting of the ordered *max* keys, that we will call the *main chain* in analogy with the FJ algorithm description[1], and the *pend* keys. The $\lfloor n/4 \rfloor$ pending elements are inserted into the main chain, using binary insertion, in the same order as the one followed by the Ford-Johnson algorithm. In this step, when $n \bmod 4 \geq 2$ the isolated pending element is also considered being the last key to be inserted into the main chain.
4. Now, we have a structure that consists of a main chain of size $\lfloor n/2 \rfloor$ which is an ordered list containing all the maximal and the pending elements and their corresponding *ltm* and *ltp* elements[2]. Additionally, we will have one isolated *ltm* element whenever $n \bmod 4 = 3$ or $n \bmod 4 = 1$. Notice that this structure corresponds exactly to the main configuration of the FJ algorithm. In this final step, the *ltm* and *ltp* elements are inserted into the main chain, using binary insertion again as is done in the FJ algorithm.

In order to give a clear description of the ${}_4$FJ algorithm, we illustrate these steps over a set of 31 keys. After the first step we obtain the configuration given in the Figure 3.

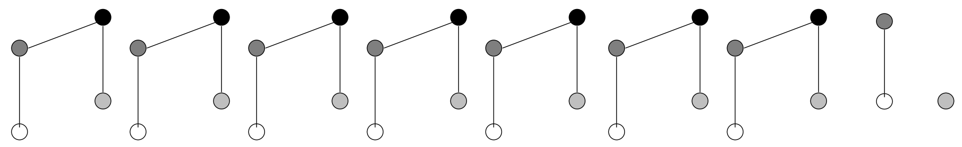
Fig. 3. ${}_4$FJ over a list of size 31. Step 1: quadruplets are built.

Then in the second step, the set of maximal elements are recursively ordered giving the configuration presented in the Figure 4.

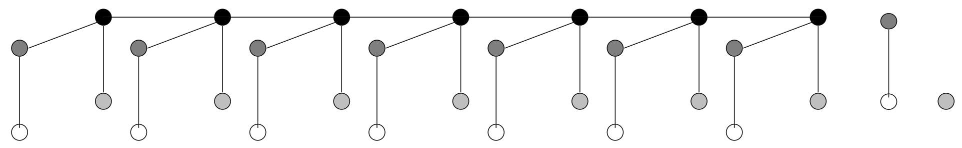
Fig. 4. ${}_4$FJ over a list of size 31. Step 2: *max* elements are recursively ordered.

[1] More precisely, the *main chain*, as defined in [4], also includes the left-most *pend* key.
[2] Again, to be more precise, the *main chain*, as defined in [4], is of size $\lfloor n/2 \rfloor + 1$ and also includes the left-most either *ltp* or *ltm* element.

In the third step, all the pending elements are inserted into the main chain of ordered maximal elements using binary insertion following the same order of the FJ algorithm. We obtain a configuration similar to the one presented in the Figure 5.

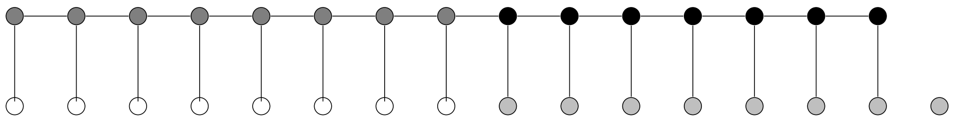

Fig. 5. ${}_4$FJ over a list of size 31. Step 3: pending elements are inserted into the main chain of maximal elements. For simplicity *max* keys where placed to the right of the *pend* keys, but they will alternate according to their relative order.

In the last configuration, the main chain consist of the *max* and *pend* elements and the "hanging" nodes are their *ltm* and *ltp* elements. Notice that for a set of 31 keys, in this point of the computation, the isolated *ltm* element has to be considered.

Finally, in the fourth step, all *ltm* and *ltp* elements are inserted into the main chain using binary insertion following the order of the FJ algorithm. We obtain the ordered list of 31 keys.

Now we will analyze the running time of the ${}_4$FJ algorithm by counting the needed number of comparisons. We use the notations presented in the previous section and in Section *5.3 Optimum Sorting* of [4]. In particular, $G(\lceil n/2\rceil)$ denotes the number of comparisons needed for inserting $\lceil n/2\rceil$ keys into a main chain of $\lfloor n/2\rfloor$ elements, using binary insertion in the order given by the FJ algorithm.

Let ${}_4F(n)$ be the worst-case number of comparisons required to sort n keys by the ${}_4$FJ algorithm. The following equation defines ${}_4F$:

$${}_4F(n) = \begin{cases} \underbrace{3\lfloor\frac{n}{4}\rfloor + \lfloor(n \bmod 4)/2\rfloor}_{\text{Step 1}} + \underbrace{{}_4F(\lfloor\frac{n}{4}\rfloor)}_{\text{Step 2}} + \\ + \underbrace{G(\lceil(2\lfloor\frac{n}{4}\rfloor + \lfloor(n \bmod 4)/2\rfloor)/2\rceil)}_{\text{Step 3}} + \underbrace{G(\lceil\frac{n}{2}\rceil)}_{\text{Step 4}} \end{cases} \tag{4}$$

This equation is explained as follows.

- The term for the first step of the ${}_4$FJ algorithm is counting three comparison for building each of the $\lfloor n/4\rfloor$ quadruplets plus one additional possible comparison whenever we have at least two remaining keys.
- The second term is counting the number of comparisons in the recursive call for ordering the maximal elements.
- The term for the third step is counting the binary insertion of the *pend* elements in the main chain of the ordered *max* elements. Notice that in the substructure involved in this step we have a total $2\lfloor n/4\rfloor$ keys plus one extra *pend* element whenever we have at least two remaining keys outside of the basic structures.
- The term for the four step is counting the binary insertion of $\lceil n/2\rceil$ keys in a main chain of $\lfloor n/2\rfloor$ elements.

Equation 4 can be simplified as

$$_4F(n) = \begin{cases} 3\lfloor \frac{n}{4} \rfloor + \lfloor (n \bmod 4)/2 \rfloor + \ _4F(\lfloor \frac{n}{4} \rfloor) \ + \\ G(\lfloor \frac{n}{4} \rfloor + \lfloor (n \bmod 4)/2 \rfloor) + G(\lceil \frac{n}{2} \rceil) \end{cases} \tag{5}$$

Since $\lfloor n/2 \rfloor = 2\lfloor n/4 \rfloor$ whenever $n \bmod 4 \leq 1$ and $\lfloor n/2 \rfloor = 2\lfloor n/4 \rfloor + 1$ whenever $n \bmod 4 \geq 1$, which implies that $3\lfloor n/4 \rfloor + \lfloor (n \bmod 4)/2 \rfloor = \lfloor n/2 \rfloor + \lfloor n/4 \rfloor$, the equation 4 can be simplified as

$$_4F(n) = \lfloor \frac{n}{2} \rfloor \ + \ \lfloor \frac{n}{4} \rfloor \ + \ _4F(\lfloor \frac{n}{4} \rfloor) \ + \ G(\lfloor \frac{n}{4} \rfloor + \lfloor (n \bmod 4)/2 \rfloor) \ + \ G(\lceil \frac{n}{2} \rceil) \tag{6}$$

Finally, observe that $\lfloor n/4 \rfloor + \lfloor (n \bmod 4)/2 \rfloor = \lceil \lfloor n/2 \rfloor /2 \rceil$. In fact, if $n \bmod 4 \geq 2$, $\lceil \lfloor n/2 \rfloor /2 \rceil = \lfloor n/4 \rfloor + 1$, which coincides with $\lfloor n/4 \rfloor + \lfloor (n \bmod 4)/2 \rfloor$ since $\lfloor (n \bmod 4)/2 \rfloor = 1$; and if $n \bmod 4 \leq 1$, $\lceil \lfloor n/2 \rfloor /2 \rceil = \lfloor n/4 \rfloor$, which implies that $\lfloor (n \bmod 4)/2 \rfloor = 0$. Thus, we obtain the following simplified expression for $_4F(n)$

$$_4F(n) = \lfloor \frac{n}{2} \rfloor + \lfloor \frac{n}{4} \rfloor + \ _4F(\lfloor \frac{n}{4} \rfloor) + G(\lceil \lfloor \frac{n}{2} \rfloor /2 \rceil) + G(\lceil \frac{n}{2} \rceil) \tag{7}$$

4 Comparing the Ford-Johnson and $_4$FJ algorithms

4.1 Running time: number of comparisons

Now, we will compare $_4F(n)$ and $F(n)$; i.e., equations 7 and 1. Inductively, we will prove that $_4F(n)$ and $F(n)$ coincide.

Initially, we expand $F(n)$ (equation 1) obtaining

$$F(n) = \lfloor \tfrac{n}{2} \rfloor + \underbrace{\lfloor \lfloor \frac{n}{2} \rfloor /2 \rfloor + F(\lfloor \lfloor \frac{n}{2} \rfloor /2 \rfloor) + G(\lceil \lfloor \frac{n}{2} \rfloor /2 \rceil)}_{= F(\lfloor \frac{n}{2} \rfloor)} + G(\lceil \tfrac{n}{2} \rceil) \tag{8}$$

which can be written as

$$F(n) = \lfloor \frac{n}{2} \rfloor + \lfloor \frac{n}{4} \rfloor + F(\lfloor \frac{n}{4} \rfloor) + G(\lceil \lfloor \frac{n}{2} \rfloor /2 \rceil) + G(\lceil \frac{n}{2} \rceil) \tag{9}$$

By induction, supposing $F(\lfloor n/4 \rfloor) = \ _4F(\lfloor n/4 \rfloor)$, equaling equations 7 and 9 we can conclude the equality of $_4F(n)$ and $F(n)$.

4.2 Recursive bookkeeping: administrative memory

Differently from the Ford-Johnson algorithm, most of the work done by the $_4$FJ algorithm is executed locally to each recursive environment originated by a recursive call. In fact, in an invocation of the $_4$FJ quadruplets are built over local data structures and a recursive call over a list of size a quarter of the initial size is done. Then for a list of size n, $_4$FJ is recursively invoked over one list of size $\lfloor n/4 \rfloor$, one of size $\lfloor n/4^2 \rfloor$, etc. These lists of keys conform the first part of the administrative memory. The second part of the administrative memory is the one needed for maintaining references between the positions of the keys of the input lists given as argument to the recursive calls and the positions of these keys in the ordered lists returned by these recursive invocations. For this we will need only data structures such as lists of pointers of the same length as the one of the input lists. This gives a total of $\sum_{i=1}^{\lceil log_4 n\rceil -1}\lfloor n/4^i\rfloor$ extra memory for the lists of keys and the same number for the pointers. Similarly, for the FJ algorithm, we use $\sum_{i=1}^{\lceil log_2 n\rceil -1}\lfloor n/2^i\rfloor$ extra memory for the keys and the same number for the pointers for maintaining the necessary connections between the original and modified key positions during the recursive calls. For estimating the difference, suppose $n = 4^k$. Then for the ${}_4$FJ algorithm we have

$$\sum_{i=1}^{\lceil log_4 n\rceil -1}\lfloor\frac{n}{4^i}\rfloor = \sum_{j=1}^{k-1} 4^j \frac{1}{3}(4^k - 1) - 1 = \frac{1}{3}n - \frac{4}{3}$$

Similarly, for FJ we obtain

$$\sum_{i=1}^{\lceil log_2 n\rceil -1}\lfloor\frac{n}{2^i}\rfloor = \sum_{j=1}^{2k-1} 2^j = 2^{2k} - 1 - 1 = n - 2$$

Supposing that the memory used by a pointer is the same than the one used by a key, we have an administrative memory for the ${}_4$FJ of

$$2(\frac{1}{3}n - \frac{4}{3}) \text{ and, similarly, for FJ the administrative memory is } 2(n-2).$$

This means that the ${}_4$FJ algorithm needs only 33% of the administrative space that the FJ algorithm needs for maintaining the ordering correspondence in each local recursive environment and all the recursive invocations.

Notice that the use of pointers can be avoided moving explicitly the keys during the recursive calls, but this should be done coordinately for quadruplets, 4^2-tuples, etc., which increases dramatically the running time. Consequently, this alternative to reduce the administrative space should be discarded.

Additionally, notice that the number of recursive calls in ${}_4$FJ is less or equal than the half of the ones needed by the FJ algorithm: $\lfloor log_4 n\rfloor / \lfloor log_2 n\rfloor < 1/2$.

5 Conclusions and future work

Although our variant of the Ford-Johnson algorithm, the ${}_4$FJ algorithm, executes exactly the same number of comparisons than the Ford-Johnson algorithm, we believe this method is of theoretical interest because it uses less administrative memory than the FJ algorithm for maintaining and coordinating the partially computed orderings during the recursive calls.

Future work could be focused on the development of adequate data structures which allow for reasonable implementations of the ${}_4$FJ algorithm. This is of importance not only for obtaining acceptable implementations of the ${}_4$FJ algorithm itself, but also for obtaining more practicality in implementations of other methods which apply the FJ algorithm. In fact, methods that improve the number of comparisons of the FJ algorithm, such as the ones presented in [6, 9, 7], are based on dividing the lists of keys into sublists which are firstly ordered applying the FJ algorithm and then efficiently merged for computing the whole ordered lists. In these methods the use of $_4$FJ is also possible (replacing all applications of FJ with $_4$FJ) and reducing in this way the needed administrative or bookkeeping memory used for controlling the recursive calls. Of course, additional considerations about the use of administrative memory are necessary, since merging of lists requires extra space. In [7], lists of 52 keys are split into lists of 10 and 42 keys which are sorted by applications of the FJ algorithm (with 22 and 171 comparisons, resp.) and then merged with Christen's merging algorithm [1] (with 37 comparisons, which gives a total of 230 comparisons: one less than $F(52) = {}_4F(52) = 231$). In [9], lists of size 42 and 5 are merged with 22 comparisons. Then, to sort lists of size 47, sublists of 42 and 5 keys are sorted separately by applying the FJ algorithm (with 171 and 7 comparisons, resp. which gives a total of 200 comparisons: one less than $F(47) = {}_4F(47) = 201$).

Additionally, current effort is focused on generalizing the $_4$FJ algorithm for octuplets, pairs of octuplets and so on. In fact, the philosophy of this method could be extended for giving a more general method $_{2^k}$FJ, which works with "2^k-tuplets", for $k > 0$. The cases $k = 1$ and $k = 2$ correspond to FJ and $_4$FJ.

References

1. C. Christen. Improving the Bounds on Optimal Merging. In *Proc. of the 19th Annual IEEE Conference on the Foundations of Computer Science*, pages 259–266. IEEE, 1978.
2. H. B. Demuth. *Electronic Data Sorting.* PhD thesis, Stanford University, 1956.
3. L. R. Ford Jr. and S. B. Johnson. A tournament problem. *American Mathematical Monthly*, 66(5):387–389, 1959.
4. D. E. Knuth. *Sorting and Searching*, volume Volume 3 of The Art of Computer Programming. Reading, Massachusetts: Addison-Wesley, 1973. Also, 2nd edition, 1998.
5. D. E. Knuth and E. B. Kaehler. An Experiment in Optimal Sorting. *Information Processing Letters*, 1:173–176, 1972. Reprinted as Chapter 30 in *Selected Papers on Algorithms*, D. E. Knuth, SLSI, 2000.
6. G. K. Manacher. The Ford-Johnson sorting algorithm in not optimal. *J. of the ACM*, 26(3):441–456, 1979.
7. G. K. Manacher, T. D. Bui, and T. Mai. Optimum Combinations of Sorting and Merging. *J. of the ACM*, 36(2):290–234, 1989.
8. M. Peczarski. Sorting 13 Elements Requires 34 Comparisons. In *European Symposium on Algorithms - ESA 2002*, volume 2461, pages 785–794. Springer Verlag, 2002. Superseded by New Results in Minimum-Comparison Sorting. *Algorithmica* 40(2):133-145, 2004.
9. J. Schulte Mönting. Merging of 4 or 5 elements with n elements. *Theoretical Computer Science*, 14:19–37, 1981.
10. M. B. Wells. Applications of a language for computing in combinatorics. In *Proceedings of IFIP Congress 65*, pages 497–498, 1965.