Small Complexity Gaps for Comparison-Based Sorting

Shogo Ehara¹, Kazuo Iwama^{1(✉)}, and Junichi Teruyama²

¹ Kyoto University, Kyoto, Japan

ehafib@gmail.com, iwama@kuis.kyoto-u.ac.jp

² Kwansei Gakuin University, Nishinomiya, Japan

junichi.teruyama@gmail.com

Abstract. Our problem is the *average* complexity on the number of comparisons for sorting algorithms. Its information-theoretic lower bound is $n \lg n - 1.4426n + O(\log n)$. Since it is easy to achieve the first $n \lg n$ term, so our focus is on the (negative) constant factor of the linear term. In their WADS2017 paper, Iwama and Teruyama obtained a new upper bound for this constant, which improves the existing gap by some 25%. However, since its main purpose is to have a closed formula for analysis, their algorithm itself, *(1,2)Insertion*, seems to have several possibilities for improvement. This paper focuses on such possibilities, and to this goal, we introduce a general scheme, *GMS*, for merge-based sorting algorithms, which includes, as a special case, *(1,2)Insertion*, the standard Merge Sort, the BinaryInsertion Sort, etc. We obtain a lower bound of the complexity for this model and three upper bounds for three specific *GMS*s. We cannot obtain closed forms for the quantity but can obtain an exact average for the number of comparisons up to $n = 65$ efficiently. The major finding is that inserting one or two elements to an already sorted sequence, which is the fundamental strategy of *(1,2)Insertion*, is quite powerful; we conjecture that more complicated merges, i.e., merging two sequences one of which is longer than two, would not help to seek a better performance.

1 Introduction

Sorting has probably the longest history of research and education among major topics in computer science. It is not only a popular routine that has appeared most frequently in all kinds of programs but also a research target that has received intensive studies from various different angles. One of the remarkable facts claiming its popularity as a research target is that a sorting algorithm which is purely aimed at its theoretical performance with almost nothing in its practical merit, due to Ford and Johnson [5], already appeared as early as the 1950's. As is well known, we met a lot of similar examples of “theoretically biased” algorithms later in the 70's and 80's.

Recall that a majority of existing sorting algorithms, including Bubble Sort, Quick Sort, Heap Sort, Merge Sort and Insertion Sort, are so-called comparison-based sort, in which the input can be accessed only through a comparison of two

(positions of the) numbers and only the 1-bit information of which is larger is available. Thus the number of comparisons needed to obtain a sorted sequence is a natural complexity measure which can be scaled more accurately than other measures such as computation time. Note that any sorting algorithm for n elements can be described as a binary (due to two different outcomes in the comparison) decision tree having $n!$ leaves corresponding to all different permutations of the n elements. The number of comparisons to obtain one of them is the number of nodes on the path from the root to the leaf corresponding to the sequence. Therefore we have an obvious lower bound, called the information-theoretic lower bound, for the number of comparisons: Any sorting algorithm needs

$$\lceil \lg n! \rceil \approx n \lg n - 1.4426n + O(\log n)$$

comparisons in the worst case.

If we are interested in only the major term, $n \lg n$, it is not hard to achieve it. For instance, consider the BinaryInsertion Sort that increases the length of the sorted sequence one by one using binary insertion of new items. We have $n - 1$ insertion steps and each of them consists of at most $\lceil \lg n \rceil$ comparisons and less ones for many of the entire steps. Thus our interest naturally comes to the constant factor for the linear term in n . Unfortunately, however, our knowledge on its quantity is quite limited in spite of the problem's long history; it is at most -0.91 for Merge Sort [8, 16] and the current best one is -1.32 for the MergeInsertion sort due to [5] mentioned above. We have had no progress for almost six decades!

Compared to the worst-case complexity, the *average-case complexity* seems more tractable (its information-theoretic lower bound is the same as the worst-case). In fact we have a number of better values for the constant; -1.26 for Merge sort [8], -1.38 for BinaryInsertion sort, and most recently -1.3999 for MergeInsertion sort [4]. Notice that 1.3999 is some 96.98% of 1.4426 , but there still exists a gap and seeking the exact bound for this fundamental problem should be an important research goal.

In [7], Iwama and Teruyama narrowed this small but still existing gap between 1.3999 and 1.4426 by some 25%. Note that the BinaryInsertion Sort, based on such a simple idea of repeating a binary insertion of a new item into a sorted sequence of length i , has a fairly good performance as shown above. Here the performance of a single binary insertion itself is optimal because it constitutes an optimal decision tree, D_i , whose path length differs at most by one. However, this optimality quickly disappears in the decision tree, $D_{i-1,i}$, of two consecutive insertions, one to a sequence of length $i - 1$ and the next to that of length i . Namely $D_{i-1,i}$ is the tree obtained by replacing each leaf of D_{i-1} by D_i , and its path length differs by two unless $i - 1$ or i is a power of two. Thus if i is close to a power of two, most of the paths still have the same length and the imbalance may not be very serious for the average complexity. However, if i is far from a power of two, only one half of the paths have the same length, a quarter has a length smaller by one and a quarter has a length longer by one, in the worst case.

This observation leads to the following natural idea for a possible improvement. If the current i is close to a power of two, we just use a binary insertion. If i is far from a power of two, then we use what we call a “two-element merge,” or *2Merge*. *2Merge* merges a two-element sequence (a_1, a_2) , $a_1 < a_2$, with a sorted sequence Y of length $i - 2$ to obtain a sorted sequence of length i . Of course we would like to see that the decision tree for *2Merge* is close to optimal or at least looks better than the above $D_{i-1,i}$. This is in fact possible and is the essence of the improvement obtained by [7].

Thus [7] achieves -1.4034 by combining *2Merge* and the binary insertion, and -1.4106 by further combining them with *MergeInsertion*. Note that the main purpose of [7] is to obtain a good theoretical performance by closed formulas, for which we need to avoid complication of algorithms as much as possible. Hence there remain several questions about the optimality of this new algorithm as well as its possible extensions. For instance, one would be quickly interested in the possibility of adding *3Merge* and *4Merge* and so on to the algorithm.

The main purpose of this paper is to answer these questions. To do so, we introduce a sorting scheme called *Generalized Merge Sort* or *GMS*. Recall that *Mergesort* cuts the given input numbers into two parts, the front part X and the rear part Y , sorts X and Y recursively, and merges the resulting sorted sequences X' and Y' . Here the standard *Mergesort* has the practice that (i) the size of X and Y is the same and (ii) when merging (sorted) X' and Y' , the order of comparisons of two numbers, one from X' and the other from Y' , is regular, left to right in both sequences. This seems ok for its compact implementation and several practical merits, but should limit its theoretical performance. Our *GMS* relaxes (i) and (ii), namely, the size of X and Y may be arbitrarily different and we can design the order of comparing two numbers in X and Y freely. Now the new sorting algorithm in [7], called *(1,2)Insertion*, is a *GMS* where $|X|$ is one or two and the order of the pair-wise comparisons is that of the standard binary search if $|X|$ is one, and carefully designed if $|X|$ is two but still based on a kind of binary search (in order to make analysis tractable).

Thus, *(1,2)Insertion* lacks two kinds of generality in terms of *GMS*. One is that (i) $|X|$ must be one or two and the other is that (ii) the comparison sequence when X is of length two is probably not optimal. To investigate the effect of (ii) we want to design a *GMS* whose $|X|$ is still one or two but the comparison sequence is optimal. This is indeed possible: For a given (unsorted) sequence of n numbers, a *GMS* first determines the cut position between X and Y and after recursively sorting X and Y , then it merges X and Y using its own order of pair-wise comparisons. Our *GMS*, called *Opt(1, 2)GMS*, can select a better cut position, 1 or 2, and if it is 1, the insertion use the standard binary search that is optimal and if it is 2, *Opt(1, 2)GMS* can find an optimal comparison sequence for the merge. These selections are done by constructing (implicitly) the entire decision tree optimal for the input length n .

To investigate (i), we want to design a *GMS* that can select any cut position between X and Y . But unfortunately, there is no obvious way of finding an optimal comparison sequence for merging X and Y in polynomial time. (If we can

spend exponential time, obtaining the absolutely optimal decision tree is possible by exhaustive search. In the case of above $Opt(1, 2)GMS$, “an implicit” construction of the optimal tree is possible in polynomial time thanks to the length of X .) Allowing exponential time is clearly unfair, so our idea is to abandon the absolute optimality or to seek an approximated optimality using heuristics and randomness.

For an input size n our GMS , $RandGMS$, first checks all $n - 1$ cut positions. For each cut position i , $|X| = i$ and $|Y| = n - i$, we further check all possible (roughly n^2) pair-wise comparisons of a in X' and b in Y' (recall we have no specific knowledge about X' or Y' other than they are sorted): For each a and b , we construct two posets (partially ordered sets) due to $a < b$ and $b < a$. Let P_1 be the poset for the former and P_2 for the latter. We then count the number of linear extensions for P_1 and P_2 (in polynomial time), and the pair a and b whose P_1 and P_2 have most balanced numbers of linear extensions is selected (the heuristics part). Then we move to one of the posets at random and recurse until the linear extension becomes unique. The number of recursion steps is the number of comparisons of this randomly selected path. By repeating this random traverse a large constant number of times we can estimate the average depth of the approximately optimal decision tree. We use this quantity plus the average number of comparisons for X and Y as the quality factor of the cut position, select the best cut position, recursively call GMS for X and Y , and merge X' and Y' using the pair-wise comparison order described above. The time complexity is apparently polynomial, but it is far larger than $n \log n$, as large as $O(n^5)$, a possible improvement of which would be our obvious target for future research.

We have obtained an average number of comparisons for these GMS s for up to $n = 65$. Note that this is not a simulation of the GMS using benchmark inputs but our performance evaluation program computes an “exact complexity” or the true average of comparison numbers for $n!$ input sequences in polynomial time. Through this numerical experiment, we have several findings, for instance, it turned out that cut positions 1 and 2 are probably enough and cut positions 3 and larger are unlikely to help for a better performance. Details are given in Sect. 5 with Table 1 summarizing several data for the number of comparisons.

Related Work. The worst-case complexity for the number of comparisons has also been a popular research topic. The second and third columns of Table 1 in Sect. 5 shows what is known up to now for small n 's [15]. Here, $C(n)$ is the information-theoretic lower bound above mentioned. $S(n)$ is the lower bound for comparison-based sorting algorithms which were obtained by exhaustive search using computers up to $n = 15$ [12–14, 18]. Thus it is known that the information-theoretic lower bound cannot be achieved by any comparison-based sorting for $12 \leq n \leq 15$. It is also known that the Ford-Johnson algorithm [1, 5] achieves a matching upper bound for $1 \leq n \leq 15$. So as far as comparison-based sorting is concerned, our knowledge is complete for $1 \leq n \leq 15$. However, $n = 16$ (sorting of only 16 numbers!) is still open, namely we have a gap of one between the lower and upper bounds for the minimum number of comparisons. It is also

known that Ford-Johnson is not optimal for some values of n larger than 16, for instance $n = 47$ [9–11].

2 Lower Bounds

A GMS is given as Algorithm 1. *GCut* computes the best cut position from solely n , the length of the unsorted sequence, without using any comparison (we conjecture this restriction does not lose generality, but do not have a formal proof). *GMerge* merges two sorted sequences (see the next section for details).

Algorithm 1. $GMS(S)$

Input: A sequence S of integers

Output: The sorted sequence S' for S

```

1  $n := |S|$ 
2 if  $n = 1$  then
3   return  $S$ 
4 end
5  $n_1 = GCut(n)$ 
6 return  $GMerge(GMS(S(1, n_1)), GMS(S(n_1 + 1, n)))$ 
```

Let $T(n)$ be the (average) number of comparisons of $GMS(S)$ for $n = |S|$. According to our assumption that no comparisons are used in *GCut*, $T(n)$ is the sum of $T(n_1)$, $T(n - n_1)$ and the number of comparisons executed in *GMerge*. Note that *GMerge* outputs a sorted sequence from two sorted sequences X and Y of length n_1 and $n - n_1$, respectively. So for fixed X and Y , there are $\binom{n}{n_1}$ different sequences as the output of *GMerge*. It then turns out that the optimal decision tree (if any) for *GMerge* that minimizes the (both average-case and worst-case) number of comparisons, is the optimal binary tree having $\binom{n}{n_1}$ leaves. Thus we have

$$T(n) \geq \min_{1 \leq i \leq \lfloor n/2 \rfloor} \left(T(i) + T(n - i) + AvDpth \left(\binom{n}{i} \right) \right),$$

where $AvDpth(M)$ is the average path length of the optimal binary tree having M leaves. The lower bound of $T(n)$ that is given by the same formula with replacing $\geq$ with $=$ can be efficiently computed from small to large n 's, its value up to $n = 65$ is given in Table 1.

3 Upper Bounds

3.1 Partially Ordered Sets

In this section, a partially ordered set (poset) plays an important role. Our posets in this paper have a special form, namely as shown in Fig. 1, they consist of two totally ordered sets and several (ordered) edges between them. The two ordered

Fig. 1. Partially ordered sets

sets correspond to two sorted sequences X and Y and edges between them are due to the results of pair-wise comparisons executed in the (generalized) merge procedure. The poset, say P_0 , in Fig. 1 (i) has X of length 4 and Y of length 8 with no comparison between them yet. Suppose we compare 3 and c and the result is $3 < c$. Then the poset changes to P_1 in (ii). If we further compare d and 7 (the result is $d < 7$), and then d and 5 (the result is $d > 5$), the poset changes to P_2 in (iii) and then P_3 in (iv). Now let us consider our knowledge about the right position of c in Y . In P_0 , we have no information about that, namely $-\infty < c < +\infty$. In P_1 , its range is narrowed to $3 < c < +\infty$. Since $c < d$, we also have $3 < d < +\infty$. Thus we increase our knowledge on the position of each item in X ; we have $-\infty < a < 7$, $-\infty < b < 7$, $3 < c < 7$ and $5 < d < 7$ in P_3 .

We denote this information by $PI(P)$ (*poset information*) for a poset P and describe it as a tuple of eight (two times the length of X) items $(x_1, x_2, \dots, x_8)$ where x_i is an item in Y or $\pm\infty$. For the poset P_3 , $PI(P_3) = (-\infty, 7, -\infty, 7, 3, 7, 5, 7)$, which is simply a combination of inequalities $3 < c$ and $3 < d$ for edge 3 to c , $5 < d$ for edge 5 to d , and $a < 7, b < 7, c < 7$ and $d < 7$ for edge d to 7. Thus it is straightforward to obtain $PI(P)$ from a poset P in polynomial time; its formal algorithm may be omitted.

Let us see the importance of poset information: Suppose that the length of Y is n in general but the length of X is small, say 4 as in Fig. 1. Then the number of different posets can be exponential since we have up to n^4 different pairs for the pair-wise comparison between X and Y and many combinations of different edge directions of them can result in posets. For instance we have 2^3 patterns by

considering all different directions of the three edges between X and Y in Fig. 1 (iv) and six of them do not include a cycle and they are posets. However, the number of different poset informations is obviously $(n+1)^8$. Thus we can use a typical DP technique when handling this type of posets if X or Y is short.

A *linear extension* of a poset P is a total order of all the items in P which does not violate any inequality in P . Due to our restriction, a linear extension of our poset is a “shuffle” of X and Y having no violation of inequalities given by the edges between X and Y . The number of different linear extensions of poset P is denoted by $L(P)$. If $PI(P_1) = PI(P_2)$, then the sets of linear extensions for P_1 and P_2 are obviously identical, and we have $L(P_1) = L(P_2)$, too.

3.2 Optimal (1,2)-Cut

A specific GMS is obtained by giving specific *GCut* and *GMerge*. We consider three different GMSs in this paper and the first one, *Opt(1, 2)GMS*, is a modified version of (1,2)Insertion in [7]. Our design is based on the construction of an optimal binary decision tree, $D(n)$, for the input size n under the condition that we follow the GMS scheme and the cut position is only 1 or 2 ($|X| = 1$ or 2), meaning *Opt(1, 2)GMS* is optimal under that condition. Suppose that we have already obtained $D(n-2)$ and $D(n-1)$. Then $D(n)$ can be obtained as follows: Let a_1 and a_2 be the first and the second element of the input (assume $a_1 < a_2$), respectively. Then $D(n)$ should be either (1) $D(n-1)$ followed by the optimal decision tree for inserting a_1 into a sorted sequence of length $n-1$ or (2) $D(n-2)$ followed by the optimal decision tree for merging (a_1, a_2) and a sorted sequence of length $n-2$. Of course $D(n)$ is the better one in terms of the average number of comparisons.

Since this decision tree has $n!$ leaves, we cannot construct it explicitly. Instead we compute, in polynomial time, the average number of comparisons, $\overline{D}(n)$, of tree $D(n)$ in a bottom-up fashion. Suppose that we have already computed $\overline{D}(n-2)$ and $\overline{D}(n-1)$. Then if we know the average number of comparisons for the merge part of (1) and (2), we can obtain $\overline{D}(n)$. The optimal decision tree for the insertion in (1) is obviously that of the standard binary search and we can compute its average number of comparisons easily. By adding this number and $\overline{D}(n-1)$, we have the average complexity of option (1).

To obtain the same quantity for option (2), we first need to design an optimal pair-wise comparison sequence for merging $X = (a_1, a_2)$ and Y of length $n-2$. This algorithm already appeared in the literature [17], which however seems quite complicated and slow. Our idea in this paper is as follows: The decision tree for this merging process is a binary tree whose node V is a poset $P(V)$. The poset $P(R)$ of the root node R consists of only two total orders corresponding to X and Y . Suppose that we first compare a and b (a is a_1 or a_2 and b is some item in Y) in the merge. Then the two sons of R , U and W , corresponding to the two outcomes of the comparison, have posets $P(U) = P(R) \cup \{a < b\}$ and $P(W) = P(R) \cup \{b < a\}$, respectively. The tree grows similarly (see Fig. 2).

In order to make this decision tree optimal, we need to select the best pair a and b among roughly $2n$ candidates (recall that $|X| = 2$ and $|Y| = n-2$) at

Fig. 2. Decision trees for merge

node R , the best pair c and d at node U and so on. To do this, we associate two numbers s and t with each node V of the tree; $s(V)$ is the number of leaves and $t(V)$ is the total path length of the subtree whose root is V . Suppose that we have already calculated $s(U), t(U), s(W)$, and $t(W)$. Then it turns out that we can compute the two values of the root R as

$$s(R) = s(U) + s(W) \text{ and } t(R) = s(U) + s(W) + t(U) + t(W).$$

Thus we can choose a and b so that $t(R)$ becomes minimum. (since the number of leaves is all the same for any pair, the minimum total path length means the minimum average length). We can obviously do this by computing s and t values of the posets appearing in the decision tree in a bottom-up fashion. Note that for each leaf V (= a poset having a unique total order) $s(V) = 1$ and $t(V) = 0$.

This is again impossible to do in polynomial time since there are exponentially many different posets even for X of size 2. Fortunately, we can bypass this problem by replacing a poset owned by each node with a poset information. As mentioned in Subsect. 3.1, the number of all different poset informations is polynomial. Now we can obtain the average complexity of option (2) by adding $t(R)/s(R)$ and $\overline{D}(n-2)$. Thus we are done with $Opt(1, 2)Cut$, which just returns the better option. It is important to know that $Opt(1, 2)Cut$ needs to compute $\overline{D}(i)$ for $1 \leq i \leq n$, since these values are exactly what we want for claiming the performance of the algorithm for a specific n . We indeed use $Opt(1, 2)Cut$ for the evaluation of our GMSs in Sect. 5. $Opt(1, 2)Merge$ just follows the comparison sequence computed by $Opt(1, 2)Cut$.

The pseudo codes are given as Algorithms 2 to 4. $VAL2M(P)$ computes the s and t values of the poset information, where we denote those values by using a subscript. $P \wedge b < a$ means the poset information for the poset P' , such that $PI(P') = P$, with the added edge from b to a . The code is described recursively, but we have to run it in a DP fashion or in a bottom-up fashion to achieve polynomial time. Recall that there are only polynomially many poset informations. $OPT(1, 2)CUT(n)$ computes the array $\overline{D}[n]$ recursively. The following routines are easy and omitted: $VALBS(n-1)$ returns the average complexity of inserting an element into a sorted sequence of length $n-1$ by the standard binary search.

BINARYINSERTION actually does this insertion. Also, OPT2MERGE merges X of length 2 and Y of length $n - 2$ following the comparison sequence determined by VAL2M.

Algorithm 2. OPT(1,2)GMS(S)

Input: A sequence S

Output: Sorted sequence of S

```

1 if  $|S| \leq 3$  then
2   return Sorted  $S$  (use any reasonable sort algorithm)
3 end
4 if OPT(1,2)CUT( $n$ ) = 1 then
5   return BINARYINSERTION( $S[1]$ , OPT(1,2)GMS( $S[2, \dots, n]$ ))
6 end
7 else
8   return
      OPT2MERGE(OPT(1,2)GMS( $S[1, 2]$ ), OPT(1,2)GMS( $S[3, \dots, n]$ ))
9 end

```

Algorithm 3. OPT(1,2)CUT(n)

Input: Integer $n \geq 4$

Output: Cut position $c = 1$ or 2, and value $v = \overline{D}[n]$

```

1  $\overline{D}[1] := 0$ ;  $\overline{D}[2] := 1$ ;  $\overline{D}[3] := 8/3$ ;
2  $\overline{D}[n - 1] := \text{OPT}(1,2)\text{CUT}_v(n - 1)$ ;  $\overline{D}[n - 2] := \text{OPT}(1,2)\text{CUT}_v(n - 2)$ 
3  $M := \text{VAL2M}_t(P_0(2, n - 2)) / \text{VAL2M}_s(P_0(2, n - 2))$ 
4 if  $\overline{D}[n - 1] + \text{VALBS}(n - 1) < \overline{D}[n - 2] + M$  then
5   return  $c = 1$  and  $\overline{D}[n] := \overline{D}[n - 1] + \text{VALBS}(n - 1)$ 
6 end
7 else
8   return  $c = 2$  and  $\overline{D}[n] := \overline{D}[n - 2] + 1 + M$ 
9   ( $P_0(2, n - 2)$  is the poset information for the poset including only two total
      orders of length 2 and  $n - 2$ .)
10 end

```

Algorithm 4. VAL2M(P)

Input: Poset information P for a poset including X , Y and arrows between them

Output: Values s and t

```

1 if  $P$  has a unique linear extension then
2   return  $s := 1$ ,  $t := 0$ 
3 end
4 Among all  $(a, b)$ 's s.t.  $a \in X$ ,  $b \in Y$  and not yet compared, find  $(a, b)$  s.t.
    $\text{VAL2M}_t(P \wedge a < b) + \text{VAL2M}_t(P \wedge b < a)$  becomes minimum
5 Let such  $(a, b)$  be  $(a_0, b_0)$ 
6 return  $s := \text{VAL2M}_s(P \wedge a_0 < b_0) + \text{VAL2M}_s(P \wedge b_0 < a_0)$  and
    $t := s + \text{VAL2M}_t(P \wedge a_0 < b_0) + \text{VAL2M}_t(P \wedge b_0 < a_0)$ 

```

3.3 Random Estimation

Our second GMS, *RandGMS*, has *RandCut* and *RandMerge* and allows us to remove the restriction of *Opt(1, 2)GMS* for which we considered only cut positions 1 and 2. Namely, *RandCut* can cut the input at any position. This is definitely an improvement over *Opt(1, 2)GMS*, but in turn we have to abandon the optimality of the decision trees as is shown in a moment. Our obvious goal is to investigate what this trade-off looks like.

The basic structure of *RandCut* is the same as *Opt(1, 2)Cut*. Again let $D(n), D(n-1), \dots, D(1)$ be the decision tree of our *RandGMS* for the input size $n, n-1, \dots, 1$. As with the previous case, $D(n)$ is obtained from $D(n-1), \dots, D(1)$. Namely, for each cut i , there is an optimal decision tree, $M(n, i)$, for merging X of length i and Y of length $n-i$. Then the best decision tree for cut i is the concatenation (each leaf is replaced by the next tree) of $D(i)$, $D(n-i)$ and $M(n, i)$. To have $D(n)$, we simply select the best i in terms of the average complexity.

Again we do not compute the trees themselves but only their average complexity, $\bar{D}(n), \bar{D}(n-1), \dots, \bar{D}(1) = 0$, in a bottom-up fashion. Suppose that we have obtained $\bar{D}(1)$ through $\bar{D}(n-1)$. Now we want to compute $\bar{D}(n)$ for the input of length n and to do so, we want to obtain a “good” cut position i . This can be done if we can evaluate $M(n, i)$ for each i namely if we can compute $\bar{M}(n, i)$ that is the average number of comparisons of $M(n, i)$. All we have to do then is to take the best $i = i_0$ that minimizes $\bar{D}(n) = \bar{M}(n, i_0) + \bar{D}(i_0) + \bar{D}(n-i_0)$. The tree $M(n, i)$ looks like the one in Fig. 2; in each node one should select the best pair to be compared and branch to the two sons due to the outcome. Again the number of different posets is exponential, but what differs from the previous section is that it does not help to replace each poset with poset information; there are exponentially many poset informations, too, because the length of X is arbitrary.

Thus we cannot do the bottom-up computation in polynomial time. What we do instead is to use an approximated value for $\bar{M}(n, i)$ or to “estimate” the true value by randomly traversing $M(n, i)$. (We abuse the same notation $\bar{M}(n, i)$ for the approximated value.) We first need to fix $M(n, i)$ by determining the pair of a in X and b in Y to be compared in the first step, by which we can determine two sons U for $a < b$ and W for $b < a$ of the root R (see Fig. 2 again). Here we would want to select the best pair minimizing the average complexity, but it is impossible as mentioned above. Instead, we use a heuristic that $s(U)$ and $s(W)$ should be balanced. (Recall that $s(U)$ and $s(W)$ the number of leaves of subtrees with root U and W , respectively.) Here $s(U)$ is equal to the number of linear extensions, $L(P(U))$, of its poset $P(U)$. This rule is the same in all the nodes of the tree. Fortunately the number of linear extensions for our type of poset can be computed in polynomial time as explained in Sect. 4, so each traverse of the path takes polynomial time.

Thus we decide the first pair as the one which achieves the most size-balanced subtrees. Then we go to one of the two sons at random, where we go to U with probability $L(P(U))/L(P(R))$ and to W with $L(P(W))/L(P(R))$ (both are close to $1/2$ but might not be exactly $1/2$). Note that $L(P(R)) = L(P(U)) + L(P(W))$.

Suppose that the random selection was U . Then we do the same thing, i.e., we select c and d to be compared next as the pair giving us the most balanced linear extensions for the two sons. We continue this until getting to a leaf and obtain the path length that is equal to the number of comparisons along this traversal of the decision tree $M(n, i)$. By repeating this traversal a (large) constant number of times, we define $\bar{Q}(n, i)$ as the average of the path length in those repeated traversals. Thus our $M(n, i)$ may not be optimal any longer and furthermore $\bar{Q}(n, i)$ is the value estimated from a small fraction of its paths.

We do this for all i 's and select the best i_0 that minimizes $D^*(n) = \bar{Q}(n, i_0) + D^*(i_0) + D^*(n - i_0)$. Since $\bar{Q}(n, i_0)$ is not exact, $D^*(n)$ is not, either. That is why we use new notations $D^*(n)$ here instead of $\bar{D}(n)$. *RandCut* just returns this best i_0 . Note that this $D^*(n)$ is used only for selecting cut positions and the sequence of comparisons for merge. As mentioned later, our performance evaluation is done using not $D^*(n)$ but $\bar{D}(n)$. See Algorithms 5 to 7 for the pseudo codes.

RandMerge simply follows the comparison sequence computed in *RandCut*, namely it always uses the pair-wise comparison that produces the most size-balanced subtrees. Once the best pair is determined, then it actually makes that comparison and goes to one of the subtrees depending on its outcome.

3.4 Combination of *Opt(1, 2)GMS* and *RandGMS*

Our $\bar{Q}(n, i)$ in *RandGMS* was an approximated value, but if $i = 1$ or 2 , we can use the exact value as described in Subsect. 3.1. Our third GMS, *OptRandGMS*, replaces this part of *RandGMS* by the same part of *Opt(1, 2)GMS*. Details may be omitted.

4 Enumeration of Linear Extensions

Enumeration of $L(P)$ for a poset P is #P-hard in general [2]. However, if the width of the poset is bounded by a constant, it can be computed in polynomial time [3]. Since all posets in this paper are even more restricted, we can use the following simple approach for its enumeration. (Similar ideas are somewhat popular for different purposes but we are not aware of any literature on its application to counting linear extensions.)

Figure 3 (i) shows a poset satisfying the restriction in this paper. We use a 2D mesh structure as illustrated in (iii) to count linear extensions. Ignore the arrow from 3 to c in the poset for a while. Then the poset has only two total orders and the corresponding mesh has complete rows and columns corresponding to items of one total order and corresponding to items of the other, respectively. A linear extension has a one-to-one correspondence to a path from the left-top corner to the right-bottom corner in the mesh. The path must follow the traversing rule, namely it can go from left to right or downward but cannot from right to left or upward. For instance, the path given by the bold line corresponds to the linear extension $1a2bcd34e56$. Now we revive the edge from 3 to c . Then this path violates this arrow because c comes before 3. Suppressing such an illegal path is easy: it suffices to delete the edge from $2b$ to $2c$ in the mesh. The reason is that

if the path comes to $2b$, then it has already passed 1 and 2 but not 3. So if the path next goes to c , then 3 must appear after that, violating the order specified by the edge from 3 to c .

Algorithm 5. RANDGMS(S)

Input: A sequence S

Output: Sorted sequence of S

```

1 if  $|S| \leq 3$  then
2   return Sorted  $S$  (use any reasonable sort algorithm)
3 end
4  $n_1 := \text{RANDCUT}(n)$ 
5 return RANDMERGE(RANDGMS( $S[1, \dots, n_1]$ ), RANDGMS( $S[n_1 + 1, \dots, n]$ ))
```

Algorithm 6. RANDCUT(n)

Input: Integer $n \geq 4$

Output: Cut position c and value $v = D^*[n]$

```

1  $D^*[1] := 0$ ;  $D^*[2] := 1$ ;  $D^*[3] := 8/3$ ;
2 for  $i = 1$  to  $n - 1$  do
3   Repeat TRAVERSE( $P_0(i, n - i)$ )  $R_0$  times and let  $\bar{Q}[i] :=$  the average of the
   returned values.  $R_0$  is a big constant.
4    $QD[i] := \bar{Q}[i] + D^*[i] + D^*[n - i]$ 
5 end
6  $c = \text{ARGMIN}(QD[i])$ 
7 return  $c$  and  $D^*[n] := QD[c]$ 
```

Algorithm 7. TRAVERSE(P)

Input: Poset information P for a poset including X , Y and arrows between them

Output: Path length l

```

1 if  $P$  has a unique linear extension then
2   return  $l := 0$ 
3 end
4 Among all  $(a, b)$ 's s.t.  $a \in X$ ,  $b \in Y$  and not yet compared, find  $(a, b)$  s.t.
 $|e_1 := \text{LEXT}(P \wedge a < b) - e_2 := \text{LEXT}(P \wedge b < a)|$  becomes minimum. ( $\text{LEXT}(P)$ 
computes  $L(P)$ ).
5 Let such  $(a, b)$  be  $(a_0, b_0)$ 
6  $r :=$  a random value in  $[0, 1]$ 
7 if  $r \leq e_1 / (e_1 + e_2)$  then
8   return TRAVERSE( $P \wedge a_0 < b_0$ ) + 1
9 end
10 else
11   return TRAVERSE( $P \wedge b_0 < a_0$ ) + 1
12 end
```

Fig. 3. Enumeration of linear extensions

For the same reason, we have to remove edges from b to c and from $1b$ to $1c$. Then due to the traversal rule, all the edges located below those removed edges should be removed, too. Thus the mesh changes as shown in the left-bottom area of (iv). We have more edges in the poset in (ii) and more edges are removed in the mesh as in (iv). Counting the number of legal linear extensions is straightforward, too. In the mesh, we put number 1 to each of the uppermost vertices and to each of leftmost vertices. For an inner vertex, we assign the value that is the sum of the value of its left-adjacent one and its upper-adjacent one. The final answer, the number of linear extensions, is the value assigned to the right-bottom corner. The algorithm obviously runs in quadratic time in the size of the poset. Details may be omitted.

5 Performance Evaluation for Small n

Table 1 shows several lower and upper bounds for the average number of comparisons, up to $n = 65$. As mentioned in Sect. 3, our three algorithms run in polynomial time. For instance, OPT(1,2)GMS computes the values of $\overline{D}[n]$, which is also the data that our performance-evaluation program wants to have and can be obviously obtained by simulating a part of OPT(1,2)GMS that computes $\overline{D}[n]$. Thus our evaluation programs also run pretty fast. The main reason for stopping at $n = 65$ is the limit of the largest integer that can be handled in the program. We did not have enough time to fix this problem, but we are sure we can in the near future. We will then be able to reach at least up to $n = 100$.

The first column $C(n)$ is the information-theoretic lower bound given in Sect. 1, which is probably not achieved by any sorting algorithm. $S(n)$ and $F(n)$ are the worst-case lower and upper bounds also described in Sect. 1. $T(n)$ is the (average-case) lower bound for GMS (see Sect. 2). The next two columns, “Opt(1,2)” and “Rand,” are upper bounds for our GMSs OPT(1,2)GMS and

RANDGMS. As is explained later, the data for our third one, OPTRANDGMS, is the same as that of OPT(1,2)GMS and omitted. Here “cut” is the cut position of the input selected by the algorithm. Recall that Opt(1,2) shows exact values, $\bar{D}[n]$ in the algorithm, namely the true average for $n!$ different sequences. Recall that RANDGMS uses a random estimation of the best cut position, but once the position is determined, the average complexity due to that decision tree can be computed exactly (thus the values in the table are different from the values in the estimated values, $D^*[n]$, in the algorithm). This needs exponential time in general, but fortunately in our case, the best estimated cut position is all 1 or 2, so this counting can be done in polynomial time, too. The next column, IT1, is the upper bound for the (1,2)Insertion in [7]. The last column, IT2, is the current best-possible upper bound that will be explained later. Here are some observations.

(1) What is most interesting and a bit surprising is that no cut position larger than 2 wins in Rand, and this is why OPTRANDGMS and OPT(1,2)GMS have the same data. Furthermore, in most n , the second best cut is still 1 or 2 (2 if the best is 1 and vice versa). Only a few cases where the second best is 4 exist: for instance, the estimated complexity (i.e., the values of $D^*[n]$) for cut 4 comes in the second place, next to cut 2, for $n = 12, 14, 21, 23, 25, 44, 46, 48$. We did not compute the real complexity (it takes much more time) for cut 4, but since the estimated values are considerably worse than those for cut 2, we guess that the exact values are also considerably worse. We can probably make a mathematical analysis to claim this interesting property by calculating the error bound of the random traversal, which should be a future research goal.

(2) Our main motivation of this paper is the numerical evaluation of (1,2)Insertion in [7]. Good news is that cut 1 and cut 2 are much more important than other cut positions, so the main idea of [7], adding cut 2, should have played an important role. It is often hard to discuss a difference of numerical data, the quick impression of IT1 data is not very bad.

(3) One demerit of random estimation is an accumulation of errors. As shown in Table 1, the cut position of Rand becomes different from that of Opt(1,2) at $n = 11$ for the first time and after that the Rand performance is always worse although the cut positions are the same.

(4) MergeInsertion [5] is surprisingly fast for some values of n . So, what if we combine this algorithm with OPT(1,2)GMS? Namely in each step we extend the selection from “cut 1 or cut 2” to “cut 1 or cut 2 or a direct execution of MergeInsertion.” The result, IT2, is extremely good, even better than the $T(n)$ lower bound (this is not a contradiction, since the new algorithm is not a GMS any longer). Note that we have no efficient evaluation algorithm for MergeInsertion, so we did it just by simulation using as many sample inputs as possible. We believe this specific algorithm is the current best one in terms of the average complexity of comparisons.

Table 1. First column is the input length $n \in \{2, \dots, 35\}$, the second column the value of $C(n)$ (the information-theoretic lower bound), the third column the worst-case lower bound, the forth column the upper bound of Ford-Johnson, the fifth column the value of $T(n)$ (the average-case lower bound for any GMSs), the 6th column the cut position realizing that lower bound. From the 7th to the 10th columns, the upper bounds for our GMSs and its cut position: the 7th and the 8th columns for $\text{OPT}(1,2)$, the 9th and the 10th columns for Rand. the 10th column (IT1) the upper bound for $(1,2)\text{Insertion}^*$ [7] and the 11th column its cut position, The 12th column (IT2) the upper bound for the combined algorithm of MergeInsertion and $(1,2)\text{Insertion}^*$ [7].

n	$C(n)$	$S(n)$	$F(n)$	$T(n)$	cut	$\text{Opt}(1,2)$	cut	Rand	cut	IT1	cut	IT2
2	1.000	1	1	1.000	1					1	-	
3	2.585	3	3	2.667	1	2.667	1	2.667	1	2.667	1	2.667
4	4.585	5	5	4.667	1	4.667	1	4.667	2	4.667	1	4.667
5	6.907	7	7	7.067	1	7.067	1	7.067	1	7.067	1	6.933
6	9.492	10	10	9.600	2	9.667	2	9.667	2	9.733	2	9.600
7	12.299	13	13	12.457	1	12.524	1	12.524	1	12.590	1	12.457
8	15.299	16	16	15.457	1	15.524	1	15.524	1	15.590	1	15.457
9	18.469	19	19	18.679	1	18.746	1	18.746	1	18.813	1	18.584
10	21.791	22	22	22.035	2	22.102	2	22.102	2	22.213	2	21.851
11	25.250	26	26	25.516	2	25.619	2	25.647	1	25.758	1	25.396
12	28.835	30 [18]	30	29.089	4	29.238	2	29.253	2	29.364	2	29.063
13	32.536	34 [12]	34	32.859	1	32.991	2	33.019	2	33.133	1	32.832
14	36.343	38 [13]	38	36.679	4	36.831	2	36.847	2	36.980	2	36.678
15	40.250	42 [14]	42	40.612	1	40.765	1	40.780	1	40.913	1	40.612
16	44.250	45/46	46	44.612	1	44.765	1	44.780	1	44.913	1	44.612
17	48.338	49/50	50	48.730	1	48.882	1	48.897	1	49.031	1	48.729
18	52.508	53/54	54	52.924	6	53.105	1	53.120	2	53.266	2	52.872
19	56.755	57/58	58	57.222	3	57.385	2	57.400	2	57.582	1	57.073
20	61.077	62	62	61.506	8	61.757	2	61.772	2	61.934	2	61.340
21	65.470	66	66	65.982	1	66.185	2	66.215	2	66.410	1	65.630
22	69.929	71 [13]	71	70.398	2	70.697	2	70.742	2	70.891	2	70.176
23	74.453	-	76	74.970	2	75.260	2	75.302	2	75.500	1	74.784
24	79.038	-	81	79.543	2	79.896	2	79.968	1	80.094	2	79.379
25	83.682	-	86	84.248	8	84.577	2	84.632	2	84.814	1	84.099
26	88.382	-	91	88.967	2	89.327	2	89.401	1	89.531	2	88.816
27	93.137	-	96	93.770	4	94.118	2	94.173	2	94.346	1	93.630
28	97.944	-	101	98.598	12	98.972	2	99.030	1	99.184	2	98.469
29	102.802	-	106	103.467	6	103.867	2	103.927	1	104.081	1	103.366
30	107.709	-	111	108.377	6	108.800	1	108.860	1	109.014	1	108.299
31	112.663	-	116	113.345	1	113.768	1	113.828	1	113.982	1	113.267
32	117.663	-	121	118.345	1	118.768	1	118.828	1	118.982	1	118.267
33	122.708	-	126	123.405	1	123.829	1	123.888	1	124.042	1	123.327
34	127.795	-	131	128.519	2	128.946	1	129.006	1	129.160	1	128.445

(continued)

Table 1. (*continued*)

n	$C(n)$	$S(n)$	$F(n)$	$T(n)$	cut	$Opt(1, 2)$	cut	$Rand$	cut	IT1	cut	IT2
35	132.924	-	136	133.684	2	134.118	1	134.177	1	134.332	1	133.616
36	138.094	-	141	138.894	2	139.329	2	139.400	1	139.546	2	138.831
37	143.304	-	146	144.087	4	144.582	2	144.665	2	144.816	1	144.020
38	148.552	-	151	149.403	1	149.872	2	149.943	2	150.094	2	149.221
39	153.837	-	156	154.705	2	155.200	2	155.302	1	155.452	1	154.435
40	159.159	-	161	160.018	20	160.561	2	160.652	2	160.792	2	159.702
41	164.517	-	166	165.419	6	165.962	2	166.086	2	166.231	1	164.960
42	169.909	-	171	170.829	2	171.395	2	171.514	2	171.637	2	170.251
43	175.335	-	177	176.285	2	176.867	2	177.025	1	177.148	1	175.762
44	180.795	-	183	181.746	2	182.368	2	182.509	2	182.618	2	181.307
45	186.286	-	189	187.250	2	187.907	2	188.086	1	188.195	1	186.885
46	191.810	-	195	192.768	2	193.475	2	193.637	2	193.729	2	192.418
47	197.365	-	201	198.356	2	199.077	2	199.275	1	199.367	1	198.057
48	202.950	-	207	203.952	2	204.704	2	204.893	2	204.959	2	203.649
49	208.564	-	213	209.614	2	210.365	2	210.587	1	210.653	1	209.343
50	214.208	-	219	215.280	2	216.050	2	216.263	2	216.307	2	214.996
51	219.881	-	225	220.973	4	221.767	2	222.005	2	222.052	1	220.742
52	225.581	-	231	226.682	4	227.509	2	227.743	2	227.768	2	226.457
53	231.309	-	237	232.475	1	233.282	2	233.528	2	233.560	1	232.250
54	237.064	-	243	238.193	8	239.078	2	239.311	2	239.339	2	238.028
55	242.845	-	249	244.029	1	244.903	2	245.154	2	245.175	1	243.865
56	248.653	-	255	249.847	6	250.748	2	250.993	2	251.015	2	249.704
57	254.485	-	261	255.722	9	256.625	2	256.870	1	256.892	1	255.581
58	260.343	-	267	261.608	2	262.522	2	262.766	1	262.788	1	261.478
59	266.226	-	273	267.480	22	268.437	1	268.682	1	268.704	1	267.393
60	272.133	-	279	273.414	1	274.370	1	274.615	1	274.637	1	273.326
61	278.064	-	285	279.361	2	280.321	1	280.566	1	280.588	1	279.277
62	284.018	-	291	285.329	1	286.289	1	286.534	1	286.556	1	285.245
63	289.995	-	297	291.313	2	292.273	1	292.518	1	292.540	1	291.229
64	295.995	-	303	297.313	1	298.273	1	298.518	1	298.540	1	297.229
65	302.018	-	309	303.343	1	304.304	1	304.548	1	304.570	1	303.260

References

1. Ayala-Rincón, M., De Abreu, B.T., De Siqueira, J.: A variant of the Ford-Johnson algorithm that is more space efficient. *Inf. Process. Lett.* **102**(5), 201–207 (2007)
2. Brightwell, G., Winkler, P.: Counting linear extensions. *Order* **8**(3), 225–242 (1991)
3. Cooper, J.: Linear Extension Counting is Fixed-Parameter Tractable in Essential Width. AMS Southeastern Sectional Fall 2013. www.math.louisville.edu/~biro/ams/cooper.pdf

4. Edelkamp, S., Weiß, A.: QuickXsort: efficient sorting with $n \log n - 1.399n + o(n)$ comparisons on average. In: CSR 2014, pp. 139–152 (2014)
5. Ford, L.R., Johnson, S.M.: A tournament problem. *Am. Math. Mon.* **66**(5), 387–389 (1959)
6. Hwang, F.K., Lin, S.: Optimal merging of 2 elements with n elements. *Acta Inform.* **1**, 145–158 (1971)
7. Iwama, K., Teruyama, J.: Improved average complexity for comparison-based sorting. *Algorithms and Data Structures. LNCS*, vol. 10389, pp. 485–496. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-62127-2_41
8. Knuth, D.E.: *The Art of Computer Programming. Sorting and Searching*, vol. 3, 2nd edn. Addison Wesley Longman Publishing Co., Inc., Redwood City (1998)
9. Manacher, G.K.: The Ford-Johnson algorithm is not optimal. *J. Assoc. Comput. Mach.* **26**, 441–456 (1979)
10. Manacher, G.K., Bui, T.D., Mai, T.: Optimum combinations of sorting and merging. *J. Assoc. Comput. Mach.* **36**, 290–334 (1989)
11. Mönting, J.S.: Merging of 4 or 5 elements with n elements. *Theor. Comput. Sci.* **14**, 19–37 (1981)
12. Peczarski, M.: Sorting 13 elements requires 34 comparisons. In: Möhring, R., Raman, R. (eds.) *ESA 2002. LNCS*, vol. 2461, pp. 785–794. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45749-6_68
13. Peczarski, M.: New results in minimum-comparison sorting. *Algorithmica* **40**(2), 133–145 (2004)
14. Peczarski, M.: The Ford-Johnson algorithm still unbeaten for less than 47 elements. *Inf. Process. Lett.* **101**(3), 126–128 (2007)
15. Peczarski, M.: Towards Optimal Sorting of 16 Elements. *arXiv preprint [arXiv:1108.0866](https://arxiv.org/abs/1108.0866)* (2011)
16. Steinhaus, H.: *Mathematical Snapshots*, pp. 37–40, New Nork (1950)
17. Thanh, M., Alagar, V.S., Bui, T.D.: Optimal expected-time algorithms for merging. *J. Algorithms* **7**(3), 341–357 (1986)
18. Wells, M.: Applications of a language for computing in combinatorics. In: *Proceedings of 1965 IFIP Congress*, North-Holland, Amsterdam, pp. 497–498 (1966)