Sorting with a Forklift

M. H. Albert and M. D. Atkinson

Department of Computer Science, University of Otago

Abstract. A fork stack is a stack that allows pushes and pops of several items at a time. An algorithm to determine which sequences of input streams can be sorted by a fork stack is given. The minimal unsortable sequences are found (there are a finite number only). The results are extended to fork stacks where there are bounds on how many items can be pushed and popped at one time. Some enumeration results for the number of sortable sequences are given.

1 Introduction

A standard analogy for explaining the operation of a stack is to speak about stacks of plates, allowing one plate to be added to, or removed from, the top of the stack. At this point a disruptive member of the audience generally asks why it is not possible to move more than one plate at a time from the top of a stack. In this paper we address his concerns.

In particular we will consider the problem of a dishwasher and his helper. The dishwasher receives dirty plates, washes them, and adds them one at a time to a stack to be put away. The helper can remove plates from the stack, but she can move more than one plate at a time. It so happens, that all the plates are of slightly differing sizes, and her objective is to make sure that when they are placed in the cupboard, they range in order from biggest at the bottom, to smallest at the top. It is easy to see that if the dishwasher processes three dishes in order: middle, smallest, largest then his helper can easily succeed in the model given (simply waiting for the largest plate, and then moving the two smaller ones on top as a pair). However, if she is replaced by a small child who can only move a single plate at a time, then the desired order cannot be achieved.

The characterisation of those permutations that can be sorted using the standard single push and pop operations of a stack is a classical result found in [6]. Similar characterisations for sorting using other data structures are considered in [2], [7], and [9] among others. In this paper we will consider the corresponding problems when we allow multiple additions to, or removals from the stack according to the analogy above.

An alternative, slightly more general, analogy provides the source of our title. We begin with a stack of boxes, called the input, labelled 1 through n in some order. We have at our disposal a powerful forklift which can remove any segment of boxes from the top of the stack, and move it to the top of another stack, the working stack. From there another forklift can move the boxes to a final, output, stack. Physical limitations, or union rules, prevent boxes being moved from the

working stack to the input, or from the output to the working stack. The desired outcome is that the output should be ordered with box number 1 on top, then 2, then 3, ..., with box n at the bottom. An example of a sorting procedure in progress is shown in Figure 1.

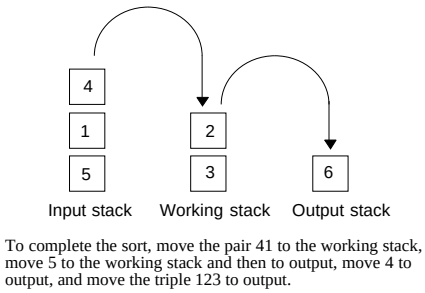

Fig. 1. A snapshot of sorting

The problems we wish to consider in this context are:

- How should such permutations be sorted?
- Which permutations can be successfully sorted?
- How many such permutations are there?

We will also consider these questions in the restricted context where one or both of the moves allowed are of limited power – for example, say, at most three boxes at a time can be moved from the input stack to the working stack, and at most six from the working stack to the output stack. The case where both moves are restricted to a single box, is, or course, the problem discussed above of sorting a permutation using a stack. We will omit most proofs entirely, or give a brief discussion of them – they will appear in subsequent papers.

The process of sorting 236415 is documented below. Note that, at the stage shown in Figure 1 it is essential that 41 be moved as a pair – moving either 4 alone, or the triple 415 would result (eventually) in 1 lying on top of 4 or 5 in the working stack, and thereby prevent sorting.

Input	Working	Output	
236415			
6415	23		
415	623		
415	23	6	(See Figure 1)
5	4123	6	
	54123	6	
	4123	56	
	123	456	
		123456	Finished

Some permutations, such as 35142 cannot be sorted. Here, we may move 3 to the working stack, and then 5 to the output, but now whether we move 1 alone, 14, or 142, we wind up with 1 lying on top of 3 or 4 in the working stack, and cannot complete the sorting procedure. We will see below that if we can avoid creating this type of obstruction in the working stack, then sorting is possible.

2 Definitions and formalities

In the subsequent sections we will tend to continue to use the terminology of the introduction speaking of the input stack, forklifts, etc. However, it will be convenient to introduce a certain amount of basic notation in order to facilitate discussion. As we will always take the initial input to be a permutation of 1 through n for some n, the contents of each stack at any time can and will be represented by sequences of natural numbers (not containing repetitions). Our ultimate objective is always to reach a state where the contents of the output stack are the permutation

$$1\,2\cdots(n-1)\,n$$

and we will refer to this outcome as success.

In the basic situation where both forklifts are of unlimited capacity, we use $\mathcal{F}$ to denote the collection of all permutations for which success is possible. If the input to working stack forklift is limited to moving j boxes in a single move, and the working to output one to moving k, then we denote the corresponding class $\mathcal{F}(j,k)$. Here j and k are either natural numbers, or ∞. We do not concern ourselves with the case where either of the forklifts is broken and incapable of making any moves!

Given a permutation π as input, a sequence of operations is *allowed*, if it does not result in an output state which provides clear evidence that sorting is not being carried out. That is, a sequence of operations is allowed if at the end of the sequence the output stack contains some tail of $12\cdots n$.

Finally, in discussing the algorithms for sorting it will be useful to pretend that it is possible to move boxes directly from the input stack to the output stack – and such an operation, as well as the more normal type of output is called *direct output*. So a direct output move consists either of output from the working stack, or moving a part of the input stack to the working stack (in a single lift), and then moving exactly that set of boxes to the output stack, again in a single lift.

Just as the case where two forklifts of unit capacity corresponds to sorting with a stack, the general case we are considering corresponds to sorting with a data structure which has a more powerful set of operations than a normal stack. Namely, we are allowed to push a sequence of objects onto the top of the structure, and likewise pop such a sequence. This structure will be called a *fork stack* (and roughly it corresponds to the working stack in our analogy).

3 The sorting algorithms

How should a fork stack actually carry out its task of sorting a permutation when this is possible? It turns out that there is a straightforward algorithm to accomplish this operation. Broadly speaking, we may use a simple modification of a greedy algorithm:

- perform any output as soon as possible,
- otherwise move the maximum decreasing sequence from the head of the input onto the working stack.

Before we justify this claim (and make some technical changes to the second option) we need a slightly more abstract characterisation of unsortability.

Definition 1. *For positive integers a and b, $a << b$ means $a < b - 1$. In a series of fork stack moves, we say that the* dreaded 13 *occurs if at some point the working stack contains adjacent elements ab with $a << b$.*

Proposition 2. *A permutation π is unsortable if and only if every allowable sequence of fork stack operations that empties the input produces, at some point, the dreaded 13.*

This is easily justified. If we cannot avoid producing a 13, then we cannot sort π for there is no way to insert the missing elements into the gap between the elements $a << b$ witnessing the 13. On the other hand, if there is some allowable sequence of operations that empties the input stack and avoids producing a 13, then on completing them, the contents of the working stack will be a *decreasing sequence, except possibly for some blocks of consecutive increasing elements.* Such a stack is easily moved to the output in its sorted order.

We refer to a sequence of the type emphasised above, as a *near-decreasing* sequence. Suppose that no immediate output is possible and consider the maximal near-decreasing sequence at the top of the input stack. If the symbols occurring in this sequence do not form an interval, then it is easy to see that any move other than taking the whole sequence and transferring it to the top of the working stack will eventually cause the dreaded 13. If the symbols occurring in the sequence do form a consecutive interval, then we can arrange to place them on the working stack in order, with largest deepest (to subsequently be moved as a block to the output). This is preferable to any other arrangement on the working stack, for it makes the top element of the working stack as small as possible, minimising the possibility of later creating a dreaded 13.

Doing direct output as soon as it becomes available can never interfere with sorting. For if we have a successful sequence of sorting moves which we modify by doing some direct output earlier, we can simply continue to carry out the successful sequence, ignoring any effect on symbols which have already been moved to output – and we will still succeed. So we may assume that any sorting algorithm does in fact perform direct output whenever it can. Then the observations of the preceding paragraph imply that when direct output is not available, the

maximal near-decreasing sequence at the top of the input stack must be moved. If this sequence contains gaps, there is no choice in how to move it, and we have argued that if it does not, then moving it so that it forms an increasing sequence on the working stack is at least as effective as any other choice.

This establishes that Algorithm 1 will correctly sort any input stack, if it is sortable at all:

Algorithm 1 Sorting with a powerful fork-lift

repeat
  Perform as many direct output moves as possible.
  Move the maximal near-decreasing sequence from the top of the input stack to the working stack, as a block if it contains gaps, so that it becomes increasing if it does not.
until input stack is empty
if working stack is empty **then**
  Success!
else
  Failure.
end if

How does Algorithm 1 need to be modified in the case where either or both of the forklifts moving from input to working stack, or from working stack to output, are of limited power? The first issue is how to modify Proposition 2. The 13 configuration is bad regardless of the power of our forklifts, but if our output lift is limited to moving k boxes we must add the condition that the working stack should not contain an increasing sequence of length longer than k. Now modifying the algorithm is straightforward. In the case where the maximal near-decreasing sequence contains gaps it must be moved as a block to avoid 13's. So, if this block is larger than the capacity of our working forklift, we fail. In the non-gap case, we would normally attempt to make the sequence increasing. Of course this would be foolish if it overwhelmed the capacity of our output lift (and it could be impossible depending on the capacity of our input lift). The only other choice that does not create a 13 is to make it decreasing, so this should be attempted if the first choice is unavailable. Failure may later occur because we create a block that is too long to move in the working stack, or a 13 there, but if not, then the algorithm will succeed.

4 Finite basis results

We now begin our combinatorial investigation of the collections of permutations sortable by various combinations of forklifts. The problem which we address in this section is how to identify the sortable or unsortable permutations without reference to Algorithm 1. In the following section we will consider the problem

of enumerating these classes. For identification purposes we concentrate on producing a list of minimal unsortable permutations. We must first prepare the way with some definitions and notation:

Definition 3. *Given permutations σ and π, we say that σ is* involved in *π, and write $\sigma \preceq \pi$ if some subsequence of π, of the same length as σ, consists of elements whose relative order agrees with those of the corresponding elements of σ. A collection of permutations closed downwards under $\preceq$ is called a* closed class.

It is easy to see that each of the collections $\mathcal{F}(j,k)$ of sortable permutations for a particular combination of forklifts is a closed class. This is because we may sort any subsequence of a sortable sequence by simply ignoring any moves that do not affect members of the subsequence. This policy cannot increase the load on a forklift in any single move, so it still sorts the remaining elements. It follows, that if we take $U(j,k)$ to be the set of $\preceq$-minimal unsortable permutations then:

$$\pi \text{ is } (j,k)\text{-unsortable} \iff \sigma \preceq \pi \text{ for some } \sigma \in U(j,k).$$

In particular, $U(j,k)$ can be thought of as a description of $\mathcal{F}(j,k)$ (or rather of its complement, but that amounts to the same thing!) This description gains some power owing to the following result.

Proposition 4. *For any $1 \leq j,k \leq \infty$ the set $U(j,k)$ is finite.*

The detailed proof of this proposition is technical and dull. It is of course enough to show that there is some finite set of permutations such that every (j,k)-unsortable permutation involves one of them. The basic idea is to make use of the characterisation of unsortability provided by the failure of Algorithm 1 (modified as necessary for the limited power case). One considers the state of the working stack, and input, at the instant where the next move being attempted either fails completely or creates a bad configuration on the working stack. The first case corresponds to a near-decreasing part which is too long for the working lift. Obviously there are a finite number of minimal examples of such, and any occurrence of one of these prevents sorting. In the second case, the cause of the resulting bad configuration can be localised to a bounded number of elements of the original input. This gives a further finite set of obstructions, from which the desired result follows.

The actual sets $U(j,k)$ are not that difficult to compute. We may assume that $k \geq j$, since the class $\mathcal{F}(j,k)$ consists of the inverses of the elements of $\mathcal{F}(k,j)$, and inversion preserves the relation $\preceq$. For the case $k > j$, once we know $U(j,\infty)$, we obtain $U(j,k)$ simply by adding the single permutation:

$$(k+1)\,k\,(k-1)\cdots 2\,1\,(k+2),$$

and then deleting any elements of $U(j,\infty)$ in which it is involved. For $j = k$, direct computation is required.

The set $U(\infty,\infty)$ consists of the permutation 35142, together with 45 permutations of length six, and 6 of length seven. The sets $U(1,k)$ are of particular interest in connection with the next section and they are:

$$\begin{aligned} U(1,\infty) &= \{2314, 3124, 3142\} \\ U(1,k) &= \{2314, 3124, 3142, (k+1)\,k\,(k-1)\cdots 2\,1\,(k+2)\} \quad (k \geq 2) \\ U(1,1) &= \{213\}. \end{aligned}$$

5 Enumeration

Associated with any collection $\mathcal{C}$ of permutations is a generating function

$$f_{\mathcal{C}} = \sum_{n=0}^{\infty} c_n x^n = c_0 + c_1 x + c_2 x^2 + \cdots$$

where

$$c_n = |\{\pi \in \mathcal{C} : \pi \in S_n\}|\,.$$

Most often one sets $c_0 = 1$, though algebraic convenience may occasionally dictate $c_0 = 0$. Recurrences which define the sequence c_n often translate naturally into algebraic or differential equations for the generating function, and indeed it is frequently more illuminating to develop these equations directly rather than via an initial recurrence (for many examples and an exposition of the theory see [5] and [10]). Additionally, having an algebraic description of the generating function, possibly only impicit, allows one to use methods based on complex analysis to provide asymptotic expansions for the numbers c_n as explained in [3] and [4]. We will pursue this program for the classes $\mathcal{F}(1,k)$.

In particular, begin with the class $\mathcal{F}(1,\infty)$. Note that the sorting algorithm is completely determined – single elements are moved from input to working stack, and output is performed whenever possible. Suppose that we have some sortable permutation π. Choose t to be the maximum integer such that the elements 1 through t occur in π in decreasing order (thus, if 2 follows 1, $t = 1$). So the original input was of the form:

$$\pi = \sigma_t\, t\, \sigma_{t-1}\, (t-1) \cdots \sigma_2\, 2\, \sigma_1\, 1\, \sigma_0$$

for some sequences σ_0 through σ_t, where $t+1$ does not occur in σ_t.

Consider the sorting procedure. The elements of σ_t are processed, and then we come to t. Now by the choice of t, $t+1$ has not yet been processed, so we may not output t (except in the trivial case where all the σ_i are empty). So we must move t to the working stack. However, if it is non-empty at this time, that move would create a 13. So the working stack must be empty, and σ_t must have been a sortable permutation of a final subinterval of the values occurring in π. Now proceed to the stage where $t-1$ is about to be moved. Again, either $t+1$ has turned up by now, and the working stack is empty, or it contains only the value t. In either case σ_{t-1} is a sortable permutation of a final subinterval of

the remaining values. This argument persists inductively. So in the end we see that $t+1$ occurs in the first non-empty σ_j, and that the general structure of a sortable permutation is:

$$(\text{sortable})\, t\, (\text{sortable})\, (t-1) \cdots (\text{sortable})\, 1\, (\text{sortable})$$

where we are free to decide the sizes of the "sortable" parts, but having done so, their elements are uniquely determined. We distinguish two cases according to whether or not the final "sortable" part is empty. By standard arguments this yields an identity satisfied by the generating function for this class which we denote f_∞:'

$$f_\infty = 1 + x f_\infty + (f-1) \sum_{t=1}^{\infty} x^t f_\infty^t$$

or, after summing the geometric series:

$$f_\infty = 1 + \frac{x f_\infty^2}{1 - x f_\infty}.$$

We can then solve the resulting quadratic to get:

$$f_\infty = \frac{1 + x - \sqrt{1 - 6x + 5x^2}}{4x - 2x^2} = \frac{2}{1 + x + \sqrt{1 - 6x + 5x^2}}.$$

The sequence that this generating function defines:

$$1,\, 1\,, 2,\, 6,\, 21,\, 79,\, 311,\, 1265,\, 5275 \ldots$$

is number A033321 in [8], and the references provided for it there connect it with other interesting enumeration problems.

The only change that needs to be made to find the generating function f_k for $\mathcal{F}(1,k)$ is to change the upper limit of summation in the relationship above from ∞ to k, since the maximum increasing sequence that we can deal with on the working stack is of length k. This allows efficient exact enumeration of these classes using standard generating function techniques. It also allows asymptotic expansions of the form:

$$c_n = \frac{r^{-n}}{n^{3/2}} \left(\sum_{k=0} \frac{e_k}{n^k} \right)$$

to be computed to any desired degree of accuracy using the methods developed in [4].

The behaviour of the radius of convergence r (whose reciprocal gives the exponential part of the growth rate for the coefficients c_n), as k increases from 1 to ∞ is particularly interesting. It begins at $1/4$, since $k=1$ gives us the Catalan numbers and then decreases to $1/5$ at $k=\infty$. However, the rate of convergence to $1/5$ is very rapid indeed. The first six values are:

$$.2500,\ .2114,\ .2033,\ .2010,\ .2003,\ .2001$$

These observations can be formalized and justified using the fact that the radius of convergence in each case is the smallest positive root of the resultant of the polynomial that f_k satisfies (whose coefficients are linear functions of x). However, the intuitive content of the result is that once you have a forklift capable of moving six boxes, you gain little in terms of which initial configurations are sortable by the addition of more power!

6 Summary and open problems

We have defined a generalised form of stack depending on two parameters j, k. It allows multiple pushes of up to j elements at a time, and multiple pops of up to k elements. For each $1 \leq j, k \leq \infty$ we have shown how to test in linear time whether an input sequence is sortable, and we have determined the minimal set of unsortable sequences.

If either $j = 1$ or $k = 1$ we have precise enumeration results for the number of sortable permutations of every length.

The first open problem to attack is the enumeration question for $j, k \geq 2$. Based on counting valid sequences of fork lift operations we have some upper bounds for these questions. Suppose that we begin with n items of input. A valid sequence of forklift operations can be represented by a path (with marked vertices) from $(0, 0)$ to $(2n, 0)$ in the plane, consisting of segments of the form (s, s) or $(p, -p)$ for positive integers s and p, representing pushing s elements onto the working stack, or popping p elements from it respectively. The condition for validity is that the path never pass below the x-axis. Figure **??** illustrates the lattice path for sorting the sequence 236415.

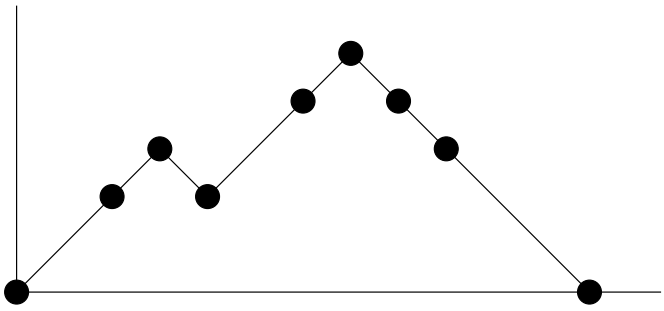

Fig. 2. The lattice path for sorting 236415

In the context of traditional stack sorting (where s and p are only allowed to equal 1) two different paths will produce two different permutations of the input. This is one of the methods for arguing that the stack-sortable permutations are enumerated by the Catalan numbers.

This same sort of analysis in fact applies to the case where only the push operation is restricted to be at most one element, and gives an alternative method of proving the enumeration results of the preceding section. However, when both s and p may take on multiple values, then it is no longer the case that two different paths produce different permutations. The simplest instance is with two input symbols 1 and 2 in that order. Then the paths:

$$(0,0) \to (2,2) \to (3,1) \to (4,0) \quad \text{and}$$
$$(0,0) \to (1,1) \to (2,2) \to (4,0)$$

both reverse the input. It is possible to reduce paths to a standard equivalent form, which reduces the problem to considering when two paths having the same outline represent the same permutation, but ambiguity is still present. None the less, by counting lattice paths we can certainly provide upper bounds for the number of sortable permutations. In particular for $j = k = \infty$, we get sequence A059231 from [8], which establishes that the number of sortable permutations of length n is $O(9^n)$.

References

1. M. D. Atkinson: Restricted permutations, Discrete Math. 195 (1999), 27–38.
2. M. D. Atkinson: Generalised stack permutations, Combinatorics, Probability and Computing 7 (1998), 239-246.
3. P. Flajolet and A. M. Odlyzko: Singularity analysis of generating functions. SIAM Jour. Disc. Math. 2 (1990), 216-240.
4. P. Flajolet and R. Sedgwick: The Average Case Analysis of Algorithms, Complex Asymptotics and Generating Functions. INRIA Research Report 2026, 1993.
5. I. P. Goulden, D. M. Jackson: *Combinatorial Enumeration*, John Wiley and Sons, New York, 1983.
6. D. E. Knuth: *Fundamental Algorithms, The Art of Computer Programming* Vol. 1 (First Edition), Addison-Wesley, Reading, Mass. (1967).
7. V. R. Pratt: Computing permutations with double-ended queues, parallel stacks and parallel queues, Proc. ACM Symp. Theory of Computing 5 (1973), 268–277.
8. N. J. A. Sloane: *The Online Encyclopedia of Integer Sequences*, `http://www.research.att.com/~njas/sequences/`, 2002.
9. R. E. Tarjan: Sorting using networks of queues and stacks, Journal of the ACM 19 (1972), 341–346.
10. H. S. Wilf: *generatingfunctionology*, Academic Press, New York, 1993.