CONGRESSUS NUMERANTIUM

WINNIPEG, CANADA

Permutations using Stacks and Queues

Edward T. Ordman
William Schmitt
Memphis State University, Memphis TN 38152 U.S.A.

Abstract

We study abstract machines that permute the integers 1 to n by storing numbers temporarily on stacks or queues. By looking at permutations using only one stack or two queues, we provide an easy one-to-one correspondence between two sets counted by the Catalan numbers. In the case of several queues, we extend a known algorithm for finding the length of an increasing sequence in a permutation to find an instance of a maximal length increasing sequence.

AMS Subject Classification: 05C, 68E

Key Words: permutation, permutation graph, stack, queue.

1 Introduction.

If the numbers $1, 2, \ldots, n$ are permuted by passing them through a channel with two parallel queues, the result is the merger of two increasing sequences. Knuth [K], (Vol 3, 5.1.4, page 64), observes that the number of such permutations is the nth Catalan number, the same as the number of permutations produced by a channel with one stack, but observes that the proof he gives is difficult and the one-to-one correspondence is not very natural (in fact, it involves work equal to sorting n numbers). We observe that if we regard the two channels as programmable devices, we can display one-to-one mappings between the sets { merger of two sequences } $\Longleftrightarrow$ { program for two-queue channel } $\Longleftrightarrow$ { program for one-stack channel } $\Longleftrightarrow$ { Catalan set, e.g. binary trees } where each correspondence is relatively straightforward, at least in the sense that each correspondence can be computed in time at worst $O(n)$. We give these correspondences in Section 2.

The situation for more than two queues, or more than one stack, is considerably more complex; it is examined at some length in [G] (Chapters 7 and 11). In section 3 we examine the situation of multiple parallel queues, showing that these techniques can be extended to provide a nice algorithm for the *upsequence* problem, that of finding longest increasing subsequences, and related problems. The upsequence problem was previously studied, for example, by [Ki].

Given the intractability of multiple-queue problems, as reported by [G], we omit discussing them further here.

We will denote a *permutation* p of the integers $(1, 2, \ldots, n)$ by $(p_1, p_2, \ldots, p_n)$. We will typically regard permutations as being produced by passing the integers $(1, 2, \ldots, n)$ through a machine which accepts integers one at a time as input, possibly stores the integers in one or more data structures, and then outputs them one at a time. For example, a machine with one stack for data storage could convert $(1, 2, 3, 4)$ into $(3, 2, 4, 1)$ by the following sequence of events: (a) (accept and) push 1; (b) (accept and) push 2; (c) (accept and) push 3; (d) pop 3 (and output it); (e) pop 2 (and output it); (f) (accept and) push 4; (g) pop 4 (and output it); (h) pop 1 (and output it). If we denote the operations push and pop by $+$ and $-$ respectively, then we can summarize this program by $+++--+--$. Such machines are considered, for example, in [K], (Vol. 1, Sec. 2.2.1, exercises and answers), and in [O].

We will use the term *permutation machine* to denote a machine such as described in the paragraph above. The specific machine there is a *one-stack permutation machine*, for obvious reasons.

The *Catalan numbers* are given by the formula $a_n = \binom{2n}{n} - \binom{2n}{n-1}$. A widely known circumstance in which these numbers occur is the number of ways in which parentheses may be introduced into a product of factors, where associativity and commutativity are not assumed. E.g. the product of two factors (aa) can be expressed in only one way, but three factors can be multiplied in two ways $((aa)a)$ and $(a(aa))$; four in five ways $((aa)(aa))$, $((a(aa))a)$, $(((aa)a)a)$, $(a(a(aa)))$, and $(a((aa)a))$; and so on. For pointers to further places these occur, see [O] or [K, Vol. 1, p.531].

Given a permutation $(p_1, p_2, \ldots, p_n)$, the *permutation graph* associated with this permutation has the vertices $1, 2, \ldots, n$ and contains an edge from p_i to p_j in exactly the cases when $i < j$ and $p_i > p_j$ in the permutation. For example, the graph determined by $(3, 4, 1, 2)$ has the edges 3-1, 3-2, 4-1, and 4-2 and is thus a quadrilateral. (We regard the edges as undirected). A graph is called a *permutation graph* if and only if it arises from some permutation in this way. For more on permutation graphs, see [G, Chapter 7].

It is well known that a permutation $(p_1, p_2, \ldots, p_n)$ is the union of exactly k (and no fewer) ascending sequences if and only if has a descending sequence of length k (and no longer one); for example $(3, 1, 4, 2)$ is the union of $(3, 4)$ and $(1, 2)$ and has no descending sequence longer than $(3, 1)$. Similarly, it is the union of exactly k descending sequences if there is a longest ascending sequence of length k. An ascending sequence is also called an *upsequence* and, if regarded as a list of vertices in the permutation graph, a maximal ascending sequence is a dominating set (every vertex in the graph is connected to it by an edge).

ACKNOWLEDGEMENTS. Conversations with C. C. Rousseau, Haklin Kimm, and R. Clapsadle have been helpful during this work.

2 One stack or two queues.

We will call a machine a *two-queue permutation machine* if it has two queues available to store numbers between receiving them and outputting them. Such a machine has available four operations: E_1 and E_2 to *enqueue* data on queue 1 and 2 respectively, and D_1 and D_2 to *dequeue* data.

We first observe that any permutation that can be produced on a two-queue machine can be produced on a simpler machine, which we call a *one-queue with bypass* machine. From now on we will abbreviate these as a 2Q-machine and a 1QB-machine, respectively. The 1QB-machine operates as follows: there is a single queue with operations E and D (enqueue and dequeue), but in addition there is a *bypass* and by the operation B an integer may be moved directly from input to output, leaving the queue unchanged. for example, the permutation (3, 4, 1, 2, 5) can be produced by the 2Q-machine by a program such as $E_1E_1E_2E_2E_2D_2D_2D_1D_1D_2$ or $E_2E_2E_1D_1E_1D_1D_2D_2E_1D_1$; it could be produced by the 1QB-machine by a program $EEBBDDED$ or $EEBBDDB$

We wish to show that this can be made general: any output of the two-queue machine can be produced by the one-queue with bypass machine. We will actually show somewhat more. Clearly every output of the two-queue machine is a union of two ascending sequences (one formed by the output of each queue). We will give an algorithm which, given a permutation that is a union of two ascending sequences, gives a program for the one-queue with bypass machine.

Algorithm 2.1 *Given a permutation $(p_1, p_2, \ldots, p_n)$ which is a union of exactly two ascending sequences, write a program for the 1QB-machine that will produce this permutation.*

We actually begin by giving an inverse algorithm: given integers arriving in the order $(p_n, p_{n-1}, \ldots, p_1)$, we produce the output $(n, n-1, \ldots, 1)$. The algorithm may be described as an output-driven, greedy algorithm. The notation below is somewhat abbreviated but should be clear.

```
input = $(p_n, p_{n-1}, \ldots, p_1)$, output = ()
nextoutput = $n$, queue = ()
WHILE nextoutput > 0 DO
    IF head_of_queue = nextoutput THEN
       BEGIN dequeue; nextoutput = nextoutput -1 END
    ELSE IF nextinput = nextoutput THEN
       BEGIN bypass; nextoutput = nextoutput -1 END
    ELSE enqueue;
OD;
```

If this algorithm is reread with the operations *dequeue, enqueue,* and *bypass* understood to include writing the characters D, E, and B to an output file, then the program will both sort the numbers in decreasing order and produce a program for the machine. For example, the sequence $(6, 5, 7, 4, 1, 3, 2)$ is placed in order

$(7,6,5,4,3,2,1)$ by the program $EEBDDBEBBD$ (6 and 5 are enqueued, 7 is bypassed, 6 and 5 are dequeued, ...).

We now simply invert the program to solve the original problem. That is, $(1,2,3,4,5,6,7)$ can be permuted into the order $(2,3,1,4,7,5,6)$ by the program $EBBDBEEBDD$; the program constructed above is written backwards with E and D interchanged.

Proposition 2.2 *The algorithm above runs in time $O(n)$ and, given as input any union of two ascending sequences, writes a program to produce it on the 1QB-machine.*

PROOF: It is clear that the program runs in time $O(n)$; it produces one token (B, D, E) per loop and produces at most $2n$ tokens. To see that the method works for any union of two ascending sequences (any product of the 2Q-machine), it suffices to observe that the integers held in the queue are precisely those that are *delayed* in the permutation; that is, k is enqueued if $k = p_j$ with $j > k$. The delayed and non-delayed elements each form an ascending sequence. The non-delayed elements are always available for bypass when needed for output; the delayed elements are always placed in the queue (some lower element is needed for output first) and are available there in the order needed since they too form an ascending sequence. □

Corollary 2.3 *The set of permutations output by the two-queue machine and by the one-queue with bypass machine are identical.*

PROOF: It is clear that any program for the one-queue with bypass machine can be translated easily for the two-queue machine. Substitute E_1 for E, D_1 for D, and E_2D_2 for B. Since the second queue is empty except when a number is actually being passed, the results are clearly the same. This, with the algorithm above, completes the proof. □

For completeness, we note that it is easy to give the same construction as the algorithm above for the one-stack machine; such a construction is implicitly given in [Knuth]. Here we state it in the "positive" order:

Algorithm 2.4 *Given a permutation which can be produced by the one-stack machine, produce a program for the one-stack machine.*

```
input = $(1,2,\ldots,n)$, wanted_output = $(p_1,p_2,\ldots,p_n)$
$k = 1$, stack = ()
WHILE $k \leq n$ DO
   IF top_of_stack = $p_k$ THEN
      BEGIN pop; $k = k + 1$ END
   ELSE push;
OD;
```

Here we suppose that *push* places the token + on the output file and *pop* places the token − on the file. Obviously, this algorithm produces a working "program" for a permutation only if the permutation can be produced by the one-stack machine. A permutation meets this condition if and only if it contains no subsequence (p_i, p_j, p_k) with $i < j < k$ and $p_i > p_k > p_j$.

We are now ready to carry out the program of one-to-one correspondences advertised in the introduction.

STEP ONE. Every permutation which is a union of two ascending sequences can be produced by the two-queue machine, or by the one-queue with bypass machine; we have produced a program for the latter machine, in time $O(n)$.

STEP TWO. We now state a one-to-one correspondence between programs produced by algorithm 2.1 and those produced by algorithm 2.4. First, note that the programs produced by algorithm 2.1 meet the following condition: the sequence ED never appears. The reason is that if the queue is empty and the next input is the desired output, we bypass the queue; if the next desired output is on the queue, we always dequeue before we enqueue. Hence E is always followed by B, never by D. Hence, we can encode 1QB-machine programs in two letters, instead of three, by substituting ED for B wherever B occurs. The original program is recoverable by substituting B for ED, so the mapping is one-to-one and clearly takes only linear time. Now we map programs using the symbols D and E to one-stack programs by mapping D to $-$ and E to $+$. The result will clearly satisfy the conditions to be a one-stack machine program (equal numbers of $+$ and $-$, and any initial substring has at least as many $+$ signs as $-$ signs), and the inverse mapping from one-stack to 1QB-machine programs meets a similar condition.

We have thus completed a proof of the following theorem.

Theorem 2.5 *The number of permutations of length n which are the union of two ascending sequences is equal to the number of sequences of length k with no subsequence* (p_i, p_j, p_k) *with* $i < j < k$ *and* $p_i > p_k > p_j$.

The resulting correspondence is not obvious to the naked eye but is easily computable by hand or in time $O(n)$. The one-to-one correspondence is pointed out in Knuth [K, vol. 3, page 64] where it is derived from a complex argument based on tableaux. Knuth states "Curiously, there appears to be no apparent way to establish a correspondence between such permutations and binary trees, except for the roundabout method via Algorithm I that we have used here." Knuth's method in dealing with unions of two ascending sequences appears to require time $O(n \log n)$; he finds the problem for the one-stack machine easy.

STEP THREE. For completeness, we go on to the correspondence between one-stack machine programs and binary trees, where we regard multiplicative notation such as $(a((aa)a))$ as a reasonable notation for binary trees.

Here is a recursive program that maps appropriate sequences of $+$ and $-$ to trees. The basic step `divide`(S, S_1, S_2) cuts S into two substrings S_1 and S_2 by finding the first place after the start of S where the number of $+$ and $-$ signs is equal; everything after that point is S_2 and what is up to that point, after dropping the leading $+$ and terminating $-$, is S_1 (So the sequence $++-++---++--$ initially splits as $S_1 = +-++--$, $S_2 = ++--$.)

```
recursive PROCEDURE maketree( S:
        sequence of + and − for 1S-machine )
    BEGIN
        IF S is empty THEN RETURN ELSE
            BEGIN divide($S, S_1, S_2$)
                PRINT '('; maketree($S_1$); PRINT ')(';
                    maketree($S_2$); PRINT ')'; RETURN;
            END;
    END;
```

For example, the sequence $++-++---++--$ maps to (() ((() ()) ())) ((() ()) ()) ; replacing each () pair by an a we get $(a((aa)a))((aa)a)$ which is the desired output. For an argument that this is a reasonable mapping between sequences of $+$ and $-$ and binary trees, see e.g. [O].

Unfortunately this program seems to require reading $O(n^2)$ symbols, since the string must be read at each level of recursion. To do things in $O(n)$ steps, we unwind the recursion and produce output as we go along. In the following program, d represents the depth of recursion (initially $d = 0$) while $h(d) = 1$ or 2 if we are processing S_1 or S_2 at that depth.

```
PROCEDURE readonce
    initially $d = 0$, $h(i) = 0$ for $i = 1 \ldots n$
    BEGIN
        READ input token
        CASE
            +:  BEGIN PRINT '('; $d = d + 1$; $h(d) = 1$; END;
            −:  BEGIN WHILE ($h(d) = 2$) DO
                    BEGIN PRINT ')';$h(d) = 0$; $d = d - 1$; END;
              IF ($h(d) = 1$) THEN
                    BEGIN PRINT ')('; $h(d) = 2$; END;
              END;
            end of input:  WHILE ($d > 0$) DO
                  BEGIN PRINT ')'; $d = d - 1$; END;
        END CASE;
    END;
```

This executes the case statement once per input token; each while loop decrements d; d is incremented only by $+$ tokens so the while loops are done at most n times for a permutation of n numbers. Thus the program executes $O(n)$ commands.

The above discussion was motivated by the following belief:

Metaconjecture 2.6 *Generally speaking, if two sequences of sets are counted by the Catalan numbers, there ought to be an easy ($O(n)$) one-to-one mapping between* *them.*

An earlier form of this belief was expressed in [O], which proposed a strategy for recursively finding one-to-one mappings between such sets. If one holds that belief, and that recursion this simple can be unwound as it was done here, then the metaconjecture becomes plausible. It is not clear how it can be formalized, if at all.

3 Several queues.

In this section we use an algorithm like 2.1 above to produce programs for machines with more than one queue. Finding maximal increasing and decreasing subsequences in a permutation and dividing it into disjoint increasing or decreasing subsequences are problems that have been examined repeatedly due to their applications in laying out VLSI chips, e.g. [DB]. Such problems are known solvable in time $O(n \log n)$, and we mainly rely on an algorithm from Chapter 7 of [G].

Algorithm 3.1 *Given a permutation $(p_1, p_2, \ldots, p_n)$, determine the lowest k such that the permutation can be produced by a k-queue machine and give the program, in time $O(n \log k)$.*

The algorithm of [G, page 167] takes a permutation as input and sorts it into ascending order. We will want one that takes the reversed permutation and sorts it into descending order, but will give merely an informal description instead of the full pseudocode.

Let k denote the next desired output; initially it is n. Let q denote the number of queues created so far; $q = 0$. If the next desired output is at the head of queue j, dequeue it and record D_j in the program. If not, then enqueue the next input. Choose a queue by finding the first existing queue $Q_1 \ldots Q_q$ whose tail is greater than the input. This involves a binary seach of the queue tails, and is the $O(\log k)$ step. Enqueue the new input there; if there is no such queue start a new one and increment q. (The queue tails will always be in increasing order.)

For example, to sort (3, 7, 6, 4, 2, 1, 5) into order (7, 6, 5, 4, 3, 2, 1), we enqueue 3 in Q_1 which for clarity we write as $E_1(3)$. The operations the algorithm produces are $E_1(3), E_2(7), D_2(7), E_2(6), D_2(6), E_2(4)$, $E_1(2), E_1(1), E_3(5), D_3(5), D_2(4), D_1(3)$, $D_1(2), D_1(1)$.

We now add one further data structure. Whenever we enqueue a number j, we store (indexed by j) a pointer to the number in the tail of the preceeding queue (if any). Thus the operation $E_2(4)$ stores a pointer from 4 to 3 (the tail of Q_1); $E_3(5)$ stores a pointer from 5 to 4. When we create a new queue we store a copy of the first entry in it in a variable named `head_of_sequence`; since in this example Q_5 is the last queue created, we will now be able to recover a sequence through the pointer chain 5, 4, 3.

The algorithm clearly, for no extra effort, divides the permutation into k descending sequences (in the above case, $(3, 2, 1)$ in queue 1, $(7, 6, 4)$ in queue 2, and (5)

in queue 3). We must show that the number of queues k is minimal. But this is clear since the sequence traced by pointers from **head_of_sequence**, when read in increasing order, is an increasing sequence of length k in the permutation.

Proposition 3.2 *Algorithm 3.1 runs in time $O(n \log k)$ and determines a program for producing the permutation; finds the length k of the longest increasing subsequence; finds an increasing subsequence of length k; and divides the partition into k decreasing subsequences.*

It is trivial to alter the algorithm to sort into ascending rather than descending order, to interchange "increasing" and "decreasing" in the above proposition. To produce a program to convert $(1, 2, \ldots, 7)$ into $(5, 1, 2, 4, 6, 7, 3)$, of course, one inverts the order of the program and interchanges D_i and E_i while preserving subscripts.

References.

[G] M. Golumbic, Algorithmic Graph Theory and Perfect Graphs, Academic Press, New York, 1980.

[K] D. E. Knuth, The Art of Computer Programming, Vol. 1, Fundamental Algorithms, 2nd. ed.; Vol. 3, Sorting and Searching, Addison-Wesley, Reading, Mass., 1973.

[Ki] H. Kimm, A modified Algorithm for the Longest Upsequence Set Problem, Proc. 31st Annual Southeast Conference, Association for Computing Machinery, Birmingham, 1993, pp. 319-322.

[DB] J. S. Deogun and B. B. Bhattacharya, Via Minimization in VLSI routing with movable terminals, IEEE Trans. Computer-Aided Design 8(1989), 917-920.

[O] E. T. Ordman, Algebraic characterization of some classical combinatorial problems, Am. Math. Monthly 78(1971) 961-970.