Rank assignment and the Robinson-Schensted-Knuth algorithm

Bill Casselman
University of British Columbia
cass@math.ubc.ca

The algorithm of Robinson and Schensted (formulated most famously by Knuth) establishes a bijection of the symmetric group $\mathfrak{S}_n$ with pairs of Young tableaux of size n and the same shape. One of the principal theorems is that if the permutation σ gives rise to the pair P, Q then σ^{-1} gives rise to Q, P. In this essay I'll present a proof of this in which this symmetry is a little more transparent than in other proofs I am aware of. What is new is that I use a problem of rank assignment in a certain partially ordered set as motivation for the algorithm. The P, Q symmetry mentioned above is a corollary of an almost obvious symmetry in rank assignment in that set.

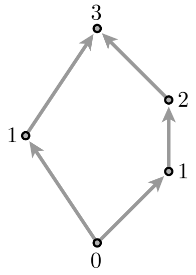
My argument is closely related to the problem that motivated Schensted's original paper, that of finding subsequences of maximal length in an arbitrary sequence of distinct integers. A natural approach to Schensted's algorithm comes from something also quite natural, the problem of assigning ranks in a partially ordered set. After a brief introduction in which I give some idea of my main result, I'll take up this theme. In this I am expanding the treatment in §4 of [Knuth:1970] (see also §5.1.4 of [Knuth:1975]).

I am very grateful to Darij Grinberg for reading earlier versions of this essay thoroughly, pointing out several places where clarity was lacking and—to my embarrassment and horror—statements incorrect.

Contents

1. Introduction

In this essay, I define the **rank** r_c of an element c in a finite partially ordered set to be the length of the longest ascending chain with terminus c. The following figure shows ranks for one example.



This note will be mainly concerned with a partial order defined on any set of integer pairs:

(1.1) $$(q,p) \le (q_*,p_*) \text{ if and only if } p \le p_* \text{ and } q \le q_* .$$

Any permutation σ in $\mathfrak{S}_n$ determines such a set, that of pairs $(m, \sigma(m))$, and one can then assign ranks to these pairs. The main result of this paper is that a suitable manipulation of these ranks allows one to deduce directly the pairs of Young tableaux corresponding to both σ and its inverse by the RSK correspondence, and to see from this that they are related by swapping.

But before I say more I need to recall that the RSK algorithm may be applied to any finite set of pairs (q_i, p_i) in which (p_i) and (q_i) are lists of n distinct integers. From such lists one gets Young tableaux P, Q, and the result about σ and σ^{-1} is a special case of the fact that if (q, p) corresponds to P, Q, then (p, q) corresponds to Q, P.

I'll now lay out the algorithm, using a single example to illustrate each step.

Step 1. *This is the start. We are going to build two tableaux P, Q, adding one row to each in each loop of the algorithm to come. At the start, each tableau is empty.*

Step 2. *This is the entry point to a loop. We have in hand a set (q_m, p_m) of pairs of integers, of the same length. It is assumed that in neither set p_m or q_m does any integer occur more than once. We are also given two tableaux P, Q.*

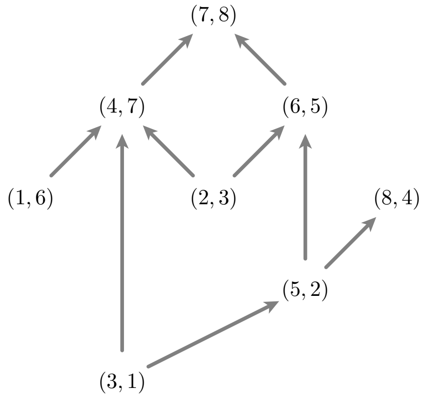
For the example, suppose the original set to be

$$(1,6), (2,3), (3,1), (4,7), (5,2), (6,5), (7,8), (8,4)\,,$$

which is the set of pairs associated to the permutation $(6, 3, 1, 7, 2, 5, 8, 4)$.

Step 3. *Compute the rank of each pair (q_m, p_m) with respect to the partial order (1.1) . List these ranks in a table, ordering all elements of the same rank according to increasing order of the first element.*

In our example, we are dealing with a very small set, for which we can easily visualize the partial order:



We can read off the ranks, to get the following table:

rank	list
0	$(1,6)$
	$(2,3)$
	$(3,1)$
1	$(4,7)$
	$(5,2)$
2	$(6,5)$
	$(8,4)$
3	$(7,8)$

In this example, *the lists are also sorted by decreasing order of the second item*. This always happens—it is because pairs of the same rank must be incomparable.

Step 4. *Transform each list of pairs of the same rank by rotating the second items down and around. Mark the first pair in each of the new lists.*

Thus

$$\begin{array}{c} (1,6) \\ (2,3) \\ (3,1) \end{array} \quad \text{becomes} \quad \begin{array}{c} (1,1) \\ (2,6) \\ (3,3) \end{array}\,,$$

and the last table becomes

rank	list
0	$(\boldsymbol{1},\boldsymbol{1})$
	$(2,6)$
	$(3,3)$
1	$(\boldsymbol{4},\boldsymbol{2})$
	$(5,7)$
2	$(\boldsymbol{6},\boldsymbol{4})$
	$(8,5)$
3	$(\boldsymbol{7},\boldsymbol{8})$

Step 5. *Strip off the top pair in each of the lists of given rank (i.e. the marked pairs in this table) and place their entries in two rows (swapping the order in each pair). Add these rows to the tops of the tableaux under construction.*

Like this:

$$(\boldsymbol{1},\boldsymbol{2},\boldsymbol{4},\boldsymbol{8}), \qquad (\boldsymbol{1},\boldsymbol{4},\boldsymbol{6},\boldsymbol{7})\,.$$

Step 6. *Make up a new list of pairs from the pairs that were not marked. If this list is not empty, loop to Step 2. Otherwise, we now have in hand two tableaux P, Q of size n and the same shape.*

In our example, the new set is

$$(2,6),(3,3),(5,7),(8,5)\,.$$

We next get in succession

rank	list
0	$(2,6)$
	$(3,3)$
1	$(5,7)$
	$(8,5)$

and, after rotating:

rank	list
0	$(\boldsymbol{2},\boldsymbol{3})$
	$(3,6)$
1	$(\boldsymbol{5},\boldsymbol{5})$
	$(8,7)$

giving us new rows

$$(\boldsymbol{3},\boldsymbol{5}), \qquad (\boldsymbol{2},\boldsymbol{5})\,.$$

The final pass is trivial, and gives us rows

$$(\boldsymbol{6},\boldsymbol{7}), \qquad (\boldsymbol{3},\boldsymbol{8})\,,$$

leading finally to the tableaux

$$\begin{array}{|c|c|c|c|}\hline 1 & 2 & 4 & 8 \\ \hline 3 & 5 \\ \cline{1-2} 6 & 7 \\ \cline{1-2}\end{array} \qquad \begin{array}{|c|c|c|c|}\hline 1 & 4 & 6 & 7 \\ \hline 2 & 5 \\ \cline{1-2} 3 & 8 \\ \cline{1-2}\end{array}$$

1.2. Theorem. *The pair of tableaux constructed this way are the same as the ones constructed by the RSK algorithm.*

Proof postponed.

What happens if we swap p and q? Let's look at the same example again. The new set is

$$(6,1),(1,3),(7,4),(2,5),(5,6),(8,7),(4,8)$$

Swapping doesn't change the rank assignments, but the ordered lists are reversed. So the new rank table is

rank	list
0	$(1,3)$
	$(3,2)$
	$(6,1)$
1	$(2,5)$
	$(7,4)$
2	$(4,8)$
	$(5,6)$
3	$(8,7)$

After rotating:

rank	list
0	$(\boldsymbol{1},\boldsymbol{1})$
	$(3,3)$
	$(6,2)$
1	$(\boldsymbol{2},\boldsymbol{4})$
	$(7,5)$
2	$(\boldsymbol{4},\boldsymbol{6})$
	$(5,8)$
3	$(\boldsymbol{8},\boldsymbol{7})$

The new rows:

$$(\boldsymbol{1},\boldsymbol{4},\boldsymbol{6},\boldsymbol{7}), \qquad (\boldsymbol{1},\boldsymbol{2},\boldsymbol{4},\boldsymbol{8})\,.$$

and the next set:

$$(6,2),(3,3),(7,5),(5,8)\,.$$

The new rows are the same as the first rows for the original set, but swapped. The new second set of pairs is the same as the original second set, but with swapped pairs!

I leave it as an exercise to show that this always happens—if we swap p, q we get Q, P instead of P, Q. We recover the classic theorem:

1.3. Corollary. *If the sequences p, q give rise to P, Q via the Schensted algorithm, then the pair q, p give rise to Q, P.*

The proof of Theorem 1.2 will take up the rest of this paper. Although the algorithm laid out above looks very simple, there is a difficulty lurking in the details—*it is not all that easy to compute ranks*. In fact, the best way to do it is by the RSK algorithm! Explaining how this works will at the same time prove the theorem. Trying to come up with a way to compute ranks in this partially ordered set will in fact lead naturally to the RSK algorithm.

2. Computing rank in a partially ordered set

I open with the general question:

How can one compute ranks in a partially ordered set?

Of course any answer must depend on how the set and its order are specified. In practice, a partial order is often specified by a list of *defining pairs* $c \prec d$. This list can then be extended to determine a partial order by defining $c \le d$ to mean there exists a chain

$$c = c_0 \prec c_1 \prec \ldots \prec c_n = d$$

of defining pairs between c and d. This works as long as the extension procedure uncovers no loops. One can expect no more, since if loops do occur there is no compatible order.

A pair $c < d$ is called **primitive** if there is no element in between them. The set of primitive pairs in a partially ordered set is the unique minimal set of defining pairs.

I say that c is a **predecessor** of d and d is a **successor** of c if (c, d) is one of the defining pairs.

So now I ask more precisely, *if a partial order is specified by defining pairs, how can ranks be calculated?*

2.1. Lemma. *The rank of an element is the maximal length of a chain of defining pairs to it.*

Proof. Any relation $x < y$ can be interpolated by a chain of defining pairs. ▯

Because of this, the following Proposition allows us to calculate ranks inductively:

2.2. Proposition. *Suppose a partial order on a set C to be defined by pairs $c \prec d$. Then*

(a) the element c has rank 0 if it has no predecessors;

(b) the element c has rank $\ell + 1$ if all of its predecessors have rank $\le \ell$ and at least one of its predecessors has rank ℓ.

The rank of an item is assigned when the ranks of *all* of its predecessors have been assigned, and cannot be done until that has been done.

This Proposition enables a practical computation of ranks by a simple variant of the algorithm for what [Knuth:1973] (in §2.2) calls **topological sorting**. In this algorithm a collection of defining pairs is given. It then arranges the elements of the set in a sequence so that $i < j$ if $c_i < c_j$, thus constructing a linear order into which the given partial order embeds. But it will also order the elements by rank and enable the ranks to be assigned.

◇ We first make an initial scan through all the input pairs, and assign each element c a pair of data: (a) the number n_c of predecessors of c encountered, and (b) a list of its successors.

◇ We now scan through all the elements in the set, assigning rank $r_c = 0$ to all the minimal ones, which are those c with $n_c = 0$. We also put them in a queue. I recall that when working with a queue, items are removed in the same order as they are put on. (It is a **F**irst **I**n, **F**irst **O**ut—i.e. FIFO—list.)

◇ We initialize the array that will eventually hold all the elements in linear order, but which starts out empty.

◇ As long as the queue isn't empty, we pop an item c off it and put it at the end of the sorted list we are building. Then we run through the list of the successors of c, decrementing the predecessor count of each. If d is one of these successors and if n_d becomes 0, then d is minimal in the current unranked group, we set $r_d = r_c + 1$, and we put it in the queue. Loop to look at the queue again.

An item is put on the queue before any of its successors, so (precisely because it *is* a queue) it is removed from it before them, and it is therefore also put on the sorted list before any of them. This ensures that the linear order implicit in the list is compatible with the original partial order.

At the end, we have both assigned ranks and listed all the elements of C in an order compatible with rank.

Along the way, we have implicitly found all primitive pairs, too, since $a < b$ is a primitive pair if and only if $r_b = r_a + 1$.

This very basic algorithm is embedded in many standard computer programs, for example in computing dependencies in the UNIX utility `make`. It is not hard to modify it slightly so as to find a chain of maximal length—when we assign r_d in the procedure above, define the *special predecessor* of d to be c. At the end, the last item in the list will have maximal rank, and going through its special predecessor, the special predecessor of this in turn, etc. builds a maximal chain.

3. Schensted's problem

The procedure laid out in the previous section will find ranks in any partially ordered set, but if the order is not given in the form of pairs $a \prec b$ we'd have to first find such data. This may not so easy to do. Anyway, we shall look at a very special class of partially ordered sets, and see for these an efficient algorithm to find ranks. This algorithm is primarily due to C. E. Schensted, although something prior was found by G. de B. Robinson. (The history is recounted on p. 60 of [Knuth:1975].)

The original problem of Schensted is to find the maximum length of an increasing subsequence of a given sequence of distinct integers. For example, consider $(6, 3, 1, 7, 2, 5, 8, 4)$. There is a very simple, if somewhat inefficient, way to solve the problem, basically reducing it to the procedure described in the previous section. First scan the array to make a list of all increasing pairs, then rescan to make a list of increasing triples that extend these, and so on. For this array, the list of pairs is

$$\begin{gathered}(6,7),\ (6,8),\ (3,7),\ (3,5),\ (3,8),\ (3,4),\ (1,7),\ (1,2),\\ (1,5),\ (1,8),\ (1,4),\ (7,8),\ (2,5),\ (2,8),\ (2,4),\ (5,8)\,.\end{gathered}$$

The set of ordered triples is then

$$(6,7,8),\ (3,7,8),\ (3,5,8),\ (1,7,8),\ (1,2,5),\ (1,2,8),\ (1,2,4),\ (1,5,8),\ (2,5,8)\,.$$

Of these, exactly one extends to a sequence of length four: $(1, 2, 5, 8)$, which extends no further.

The extra steps from pairs to triples, etc. is actually closely related to the process we went through in §2. The list of increasing pairs is a set of defining pairs for a partial order on a certain set. It can be interpreted as a list of relations

$$6 \prec 7,\ 6 \prec 8,\ 3 \prec 7,\ \ldots$$

This notation is not ideal, however! The relation $6 \prec 7$ does not mean just that $6 < 7$, but that $6 < 7$ and 6 occurs before 7 in the given sequence. This suggests that we make the original list (p_i) into a new sequence of pairs (i, p_i). The order imposed on this is that we have seen earlier:

$$(i, p) \le (i_*, p_*) \quad \text{when} \quad i \le i_* \text{ and } p \le p_*\,.$$

What is special about this partial order is that the initial sequence is ordered according to the initial entry of each pair.

3.1. Lemma. *The map taking $p_{i_1} < p_{i_2} < \ldots$ to $(i_1, p_{i_1}) < (i_2, p_{i_2}) < \ldots$ is a bijection between increasing subsequences in the original sequence and chains in the new ordered set of pairs.*

As a consequence, the length of the longest subsequence is equal to the maximum rank assigned.

Being given the sequence of the p_i means that in effect we are *starting* with a topological sort of the pairs (i, p_i). For example, for the sequence $(6, 3, 1, 7, 2, 5, 8, 4)$ we get the pairs

$$(1,6),\ (2,3),\ (3,1),\ (4,7),\ (5,2),\ (6,5),\ (7,8),\ (8,4)\,.$$

The additional initial coordinate will play a role later on, but can be forgotten for a while.

With this approach, the hardest work goes into finding all the defining pairs. The amount of effort this requires is basically proportional to the square of the size of the list. This is not an outrageous amount of work, but difficult enough, and impossible by hand on a large list, for example this one:

$$\begin{gathered}(41, 93, 31, 73, 98, 29, 12, 54, 24, 0, 52, 78, 87, 55, 25, 81, 76, \\ 91, 51, 7, 39, 92, 65, 40, 45, 5, 1, 20, 84, 99, 27, 32, 13, 8, 2, \\ 61, 19, 9, 74, 60, 66, 79, 47, 86, 30, 3, 85, 42, 89, 43, 70, 17, \\ 6, 63, 28, 11, 34, 75, 22, 64, 59, 16, 48, 15, 90, 80, 69, 67, \\ 35, 72, 50, 14, 33, 53, 10, 38, 94, 18, 58, 46, 49, 88, 68, 21, \\ 62, 44, 97, 82, 37, 83, 95, 4, 56, 57, 77, 23, 96, 26, 36, 71)\,,\end{gathered}$$

which we shall be able to handle later without difficulty.

We can now assign ranks according to the following slightly more direct process:

◇ The first item p_1 has no predecessors and consequently gets rank 0;
◇ the item p_i $(i > 1)$ gets rank h if
(a) there is a prior item $p_j < p_i$ ('prior' means $j < i$) of rank $h - 1$ and
(b) there are no prior items $p_j < p_i$ of rank $\geq h$.

That is to say, we assign ranks as we read the sequence p_i, instead of finding first all the increasing pairs. When assigning a rank to p_n the obvious thing to do is scan back through the ranks r_i of the prior p_i (i.e. with $i < n$) looking to verify (a) and (b). For example, look again at $(6, 3, 1, 7, 2, 5, 8, 4)$. We start with

p_i	6	3	1	7	2	5	8	4
r_i	0							

We have $p_2 = 3$, and since $p_1 = 6 > p_2$ the rank of this new item is also 0.

p_i	6	3	1	7	2	5	8	4
r_i	0	0						

Similarly $r_3 = 0$. What about r_4? Since $p_4 = 7 > p_1 = 6$, $p_2 = 2$, $p_3 = 1$, all of which have rank 0, we have $r_4 = 1$

p_i	6	3	1	7	2	5	8	4
r_i	0	0	0	1				

Usw. Here all the ranks have been assigned:

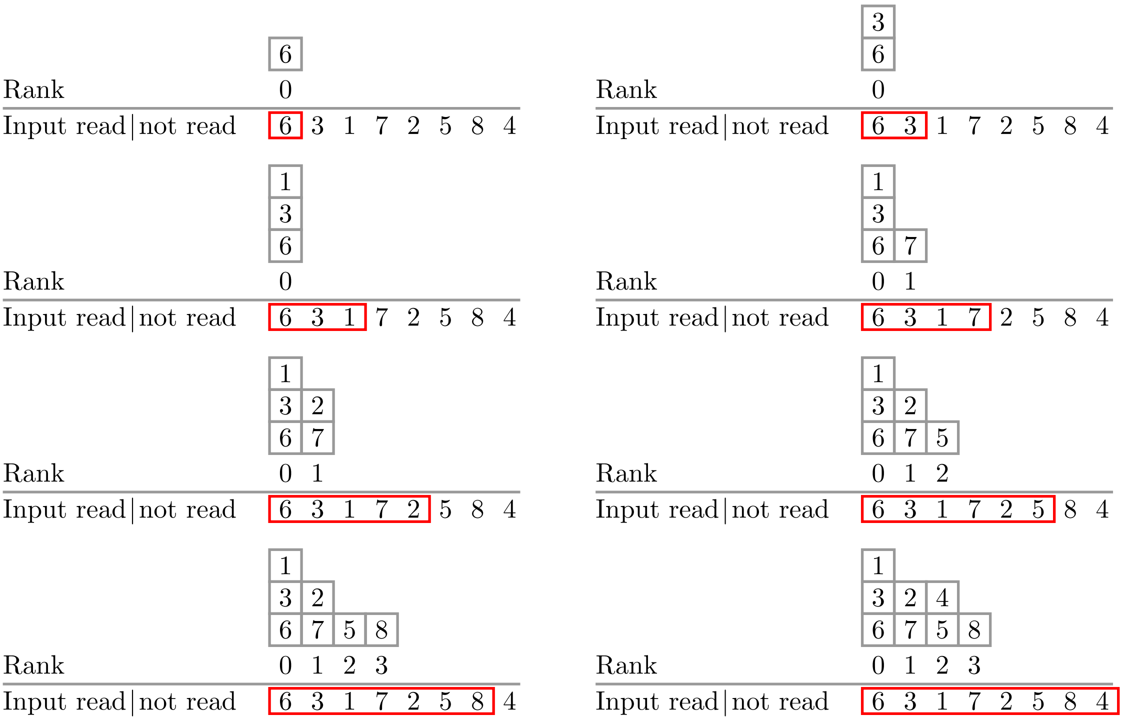
p_i	6	3	1	7	2	5	8	4
Rank	0	0	0	1	1	2	3	2

This process is still not ideal, because the amount of time it takes still seems to be roughly proportional to n^2. Some version of this procedure works for any partially ordered set, but there is something special about the one at hand that allows us to do better. As often in programs, we can gain time by using more space to record progress. In this case, however, we shall eventually be able to reclaim the space, only temporarily lost, at no cost in time.

4. Schensted's solution

There is one way to modify the procedure so as to work a bit more efficiently. It is not necessary to scan through all the prior p_i when assigning the rank of p_j. We shall maintain lists of the objects of various ranks, and in order of rank. Then in order to assign the rank of p_j, we have only to scan backwards though these lists, starting with the highest rank assigned so far, until we find one p_i with $i < j$ and $p_i < p_j$. At this point, we have located an item prior to p_j with maximum rank. We set $r_j = r_i + 1$, and add p_j to the list of items of rank r_j.

Here is how this new process goes, in a modestly more efficient but suggestive version. Above each rank is a stack of all p_i of that rank, which is expanded by adding items at the top. The row labeled 'rank' is just an index to the lists above it, and no longer records the rank of items just below in the bottom row. Instead, these ranks are recorded in the rank lists.



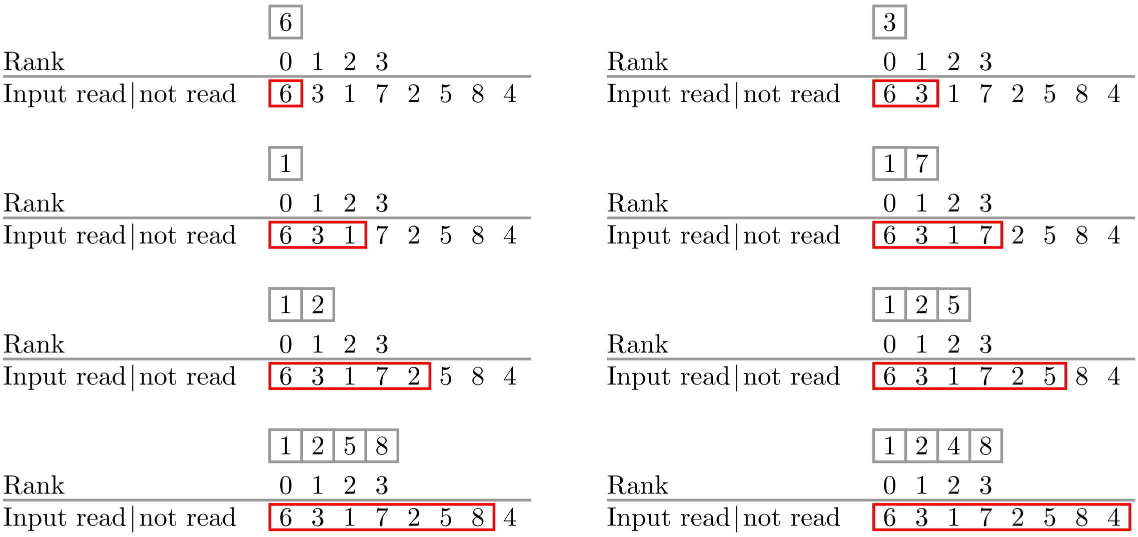
6
Rank 0
Input read|not read 6 3 1 7 2 5 8 4
3
6
Rank 0
Input read|not read 6 3 1 7 2 5 8 4
1
3
6
Rank 0
Input read|not read 6 3 1 7 2 5 8 4
1
3
6 7
Rank 0 1
Input read|not read 6 3 1 7 2 5 8 4
1
3 2
6 7
Rank 0 1
Input read|not read 6 3 1 7 2 5 8 4
1
3 2
6 7 5
Rank 0 1 2
Input read|not read 6 3 1 7 2 5 8 4
1
3 2
6 7 5 8
Rank 0 1 2 3
Input read|not read 6 3 1 7 2 5 8 4
1
3 2 4
6 7 5 8
Rank 0 1 2 3
Input read|not read 6 3 1 7 2 5 8 4

For example, in assigning rank to the final item $p_8 = 4$, we scan backwards through the rank columns until we find an entry equal to 2, with rank 1. So $r_8 = 2$, and 4 goes on top of the stack of items of rank 2.

Incidentally, in this example the array of items on top of the sequences has the shape of a Young diagram. That is to say, the number of items with rank r is a weakly decreasing function of r. That is not invariably the case, as you can see by considering the input sequence $[1, 3, 2]$.

The figures illustrate a feature that will allow an even faster process. *The columns hold more information than we need to assign ranks.* This is because *the numbers in each column of a given rank decrease as you go upwards.* If p_i and p_j have the same rank with $i < j$, then the rank of p_j is assigned later than that of p_i, and it must sit above p_i on this column. If $p_j > p_i$, then when it came to assigning r_j we would have placed it at a rank $> r_i$. Therefore the number, say p_k, at the top of each column is the least in the column. If we are assigning a rank to p_ℓ with $\ell > k$, we do not need to look at the entries beneath it, because if $p_\ell < p_k$ it is also less than those other entries.

What this means is that *we need only to keep a record of the top entries in each column.* In other words, when we assign the rank of p_ℓ, we replace the current item at the top of the column for the given rank with p_ℓ. This leads to the following sequence of figures instead of the ones above.


6
Rank 0 1 2 3
Input read|not read 6 3 1 7 2 5 8 4
3
Rank 0 1 2 3
Input read|not read 6 3 1 7 2 5 8 4
1
Rank 0 1 2 3
Input read|not read 6 3 1 7 2 5 8 4
1 7
Rank 0 1 2 3
Input read|not read 6 3 1 7 2 5 8 4
1 2
Rank 0 1 2 3
Input read|not read 6 3 1 7 2 5 8 4
1 2 5
Rank 0 1 2 3
Input read|not read 6 3 1 7 2 5 8 4
1 2 5 8
Rank 0 1 2 3
Input read|not read 6 3 1 7 2 5 8 4
1 2 4 8
Rank 0 1 2 3
Input read|not read 6 3 1 7 2 5 8 4

Now what happens when we read the last item 4? We scan back through the row of boxes, until we get blocked by the entry 2. At that point we drop 4 into the box just ahead of it, replacing the 5. It is said that the 4 **bumps** or **bounces** or **extrudes** the 5. This, finally, is Schensted's **row-insertion process**.

The sequence of 100 integers written down at the beginning is now entirely feasible. We get the length of a longest subsequence to be 15, and a subsequence of that length to be

i	9	26	34	45	52	55	71	72	75	79	80	92	93	94	95
p_i	0	1	2	3	6	11	14	33	38	46	49	56	57	77	96

There is a minor problem with this method—it no longer records permanently all rank assignments. What it does do is record the maximum height attained, which from now on we'll be satisfied with.

These figures illustrate another feature of this process: *The sequence of tops—the numbers in the boxes—are a monotonic increasing sequence, read left to right.* This can be proved by induction on the number of items in the sequence to be read.

It is not too difficult to carry out this by hand, but when programming it for large sequences you will want to be more efficient. Searching backwards through the row is important theoretically, as we'll see later, but the amount of time it takes is roughly proportional to the length of the row. You can do much better with a binary search to locate where to insert and bounce.

I'll now lay out the process more completely.

◇ We start with a sequence of distinct integers p_i. I'll assume that we want merely to say what the length of the longest increasing subsequence is—actually finding that subsequence would involve an easy modification that I have already suggested in the first section.

◇ The process maintains the array that sits at the tops of the stacks of the p_i of a given rank. The state at any moment therefore consists of two arrays—this rank array and the input sequence.

◇ At the start the first is empty and the second is the entire sequence of the p_i. In each step, the next input p_k is removed from input and the rank array is modified—either (a) an entry in the array is changed, or (b) the array is extended. (In Knuth's account, these two cases are handled as one by assuming the initial rank array to be of infinite length and with all initial entries equal to ∞.) The rank array (r_i) is scanned backwards until either (a) its beginning is encountered, in which case I set $\ell = -1$, or (b) the first r_ℓ is found satisfying the condition $r_\ell < p_k$. Then p_k is inserted as $r_{\ell+1}$. The rank array remains monotonic at all times, and if p_k is greater than all of its entries, p_k is appended to it.

5. Schensted’s extended algorithm

The process has been stripped down to an essential minimum. However, there is something more that can be done, although we might not realize its value at first. When p_j is inserted at location r_i, the old value at that location is thrown away. In the new process, we are going to *recycle* it into a new input sequence. Here is the earlier example, showing the input sequence assembled from the items to be recycled.

$\boxed{6}$	3 1 7 2 5 8 4	$\boxed{3}$	1 7 2 5 8 4 6
$\boxed{1}$	7 2 5 8 4 6 3	$\boxed{1}\boxed{7}$	2 5 8 4 6 3
$\boxed{1}\boxed{2}$	5 8 4 6 3 7	$\boxed{1}\boxed{2}\boxed{5}$	8 4 6 3 7
$\boxed{1}\boxed{2}\boxed{5}\boxed{8}$	4 6 3 7	$\boxed{1}\boxed{2}\boxed{4}\boxed{8}$	 6 3 7 5

In other words, we now see the basic process as having input a sequence of distinct integers, and as output a row of inserted numbers together with a new sequence of the extruded numbers.

What do we do with the new sequence? *We apply the Schensted process to it, but tack the new row on below the first. Etc.* In other words, we are going to use them to extend the rank array into a larger diagram. In effect, we shall assemble a new input sequence from these items, and add on arrays below the rank array to handle them. But we don’t have to perform the actual assembly, we can just insert an item as soon as it is bounced. What we get is a **tableau**, as I’ll explain in the next section. This is no longer has any obvious relationship with Schensted’s original problem, but it is nonetheless an interesting and useful thing to do. Here’s what happens in our running example (with the bounce path marked):

Step	Tableau	Remaining input
1		6 3 1 7 2 5 8 4
2	$\fcolorbox{black}{lightgray}{6}$	3 1 7 2 5 8 4
3	$\fcolorbox{black}{lightgray}{3}$ $\fcolorbox{black}{lightgray}{6}$	1 7 2 5 8 4
4	$\fcolorbox{black}{lightgray}{1}$ $\fcolorbox{black}{lightgray}{3}$ $\fcolorbox{black}{lightgray}{6}$	7 2 5 8 4
5	$\boxed{1}\fcolorbox{black}{lightgray}{7}$ $\boxed{3}$ $\boxed{6}$	2 5 8 4
6	$\boxed{1}\fcolorbox{black}{lightgray}{2}$ $\boxed{3}\fcolorbox{black}{lightgray}{7}$ $\boxed{6}$	5 8 4
7	$\boxed{1}\boxed{2}\fcolorbox{black}{lightgray}{5}$ $\boxed{3}\boxed{7}$ $\boxed{6}$	8 4
8	$\boxed{1}\boxed{2}\boxed{5}\fcolorbox{black}{lightgray}{8}$ $\boxed{3}\boxed{7}$ $\boxed{6}$	4

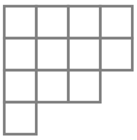
9 $\begin{array}{cccc} 1 & 2 & 4 & 8 \\ 3 & 5 & & \\ 6 & 7 & & \end{array}$

Let's look at Step #3, for example. The 3 bumps the 6, and we start a new row just below the first row with that 6. Or Step #8: the next input integer to deal with is 4. Since $2 < 4 < 5$ this bounces 5. The 5 bounces the 7 in the second row, and the 7 is finally just appended to the third row.

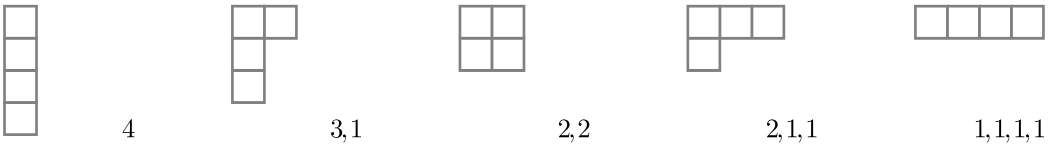
At this point, we have seen two versions of an extended Schensted process, one that assembles a new input sequence in the course of insertions, and the other that performs an immediate cascade of insertions with bounced items. The first, which I call **delayed reinsertion**, is very useful to keep in mind, because it amounts to recursion. I call the second **immediate reinsertion**.

6. Tableaux

A **Young shape** is a sequence of integers $n_1 \geq \ldots \geq n_k > 0$. This gives rise to a diagram of k arrays of rectangular boxes, with n_i boxes in row i, aligned to the left. For example, if the sequence is $[4, 4, 3, 1]$ we get the shape



Shapes with n boxes are in one-one correspondence with partitions of n. For example, there are 5 different shapes with 4 boxes:



A **(Young) tableau** is one of these sets of n empty boxes filled with a set of n distinct positive integers, satisfying the condition that entries in rows are increasing from left to right, and in columns from top to bottom. (This is sometimes called a **standard** or **strict** tableau.) To continue the example above, there are 10 Young tableaux of size 4 with set $[1, 4]$:

$\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array}$ $\begin{array}{cc} 1 & 2 \\ 3 & \\ 4 & \end{array}$ $\begin{array}{cc} 1 & 3 \\ 2 & \\ 4 & \end{array}$ $\begin{array}{cc} 1 & 4 \\ 2 & \\ 3 & \end{array}$ $\begin{array}{cc} 1 & 2 \\ 3 & 4 \end{array}$

$\begin{array}{cc} 1 & 3 \\ 2 & 4 \end{array}$ $\begin{array}{ccc} 1 & 2 & 3 \\ 4 & & \end{array}$ $\begin{array}{ccc} 1 & 2 & 4 \\ 3 & & \end{array}$ $\begin{array}{ccc} 1 & 3 & 4 \\ 2 & & \end{array}$ $\begin{array}{cccc} 1 & 2 & 3 & 4 \end{array}$

6.1. Proposition. *The result of the extended Schensted process is a tableau.*

Proof. Applying the Schensted construction by immediate recursion, it suffices to show that inserting a single item into a tableau produces a tableau. It follows from the definition of the process that rows remain monotonic increasing. What about the columns? Suppose a bumps b in row m. Then $a < b$. Suppose c in row $m + 1$ lies just below where b did, so $b < c$. If this spot is empty, think of c as ∞. In any case, b will be inserted in row $m+1$ weakly to the left of c. If it replaces c, no problem. Otherwise, all the items weakly to the left of a in row m are $< a$, so b is greater than any of them. Apply the simple recursion. □

If $p = (p_i)$ is the original sequence, I'll call P_p the tableau constructed by Schensted's construction. In my examples, the set from which the entries in a tableau are filled will usually be the range $[1, n]$, but any set of n distinct integers will do. This is important for arguments by recursion.

7. A second tableau

A single step of the extended Schensted process is **insertion** of an input integer into the tableau constructed so far. This consists of several bounces, at the end of which is a final placement. We can reverse a step if only we know where the final placement in that step occurred.

$$\begin{matrix} 1 & 2 & 5 & 8 \\ 3 & 7 & & \\ 6 & & & \end{matrix} \qquad\qquad 4$$

$$\begin{matrix} 1 & 2 & 4 & 8 \\ 3 & 5 & & \\ 6 & 7 & & \end{matrix}$$

In this example, the succession of bounces is shown in grey. The last insertion was the 7 in row 3. But once we know where it was inserted, we can deduce that the 7 was bounced from row 2 by insertion of 5, since 5 is the last entry in that row < 7; which was in turn bounced from row 1 by insertion of 4, since 4 is the last entry in that row < 5. Therefore *we can reconstruct the process triggered by that one insertion if only we record the place where this final insertion is made.*

To reconstruct the entire process we must record all of these places. We do this by means of a second tableau, which we maintain simultaneously with the first. For this, we now realize the input σ as a sequence of pairs (q_i, p_i) (with $q_i = i$ in my examples). As we go, we maintain two tableaux $P = P_\sigma$ and $Q = Q_\sigma$. The tableau P, as we have seen, is what we get from the process already described. How Q is constructed is very simple. If the final placement involved in insertion of p_i into P is at place (r, c), then we place q_i at the same place in Q. Thus:

Step	P	Q	Remaining input pairs (q, p)
1			$(1,6)\ (2,3)\ (3,1)\ (4,7)\ (5,2)\ (6,5)\ (7,8)\ (8,4)$
2	$\begin{matrix}6\end{matrix}$	$\begin{matrix}1\end{matrix}$	$(2,3)\ (3,1)\ (4,7)\ (5,2)\ (6,5)\ (7,8)\ (8,4)$
3	$\begin{matrix}3\\6\end{matrix}$	$\begin{matrix}1\\2\end{matrix}$	$(3,1)\ (4,7)\ (5,2)\ (6,5)\ (7,8)\ (8,4)$
4	$\begin{matrix}1\\3\\6\end{matrix}$	$\begin{matrix}1\\2\\3\end{matrix}$	$(4,7)\ (5,2)\ (6,5)\ (7,8)\ (8,4)$
5	$\begin{matrix}1&7\\3&\\6&\end{matrix}$	$\begin{matrix}1&4\\2&\\3&\end{matrix}$	$(5,2)\ (6,5)\ (7,8)\ (8,4)$
6	$\begin{matrix}1&2\\3&7\\6&\end{matrix}$	$\begin{matrix}1&4\\2&5\\3&\end{matrix}$	$(6,5)\ (7,8)\ (8,4)$
7	$\begin{matrix}1&2&5\\3&7&\\6&&\end{matrix}$	$\begin{matrix}1&4&6\\2&5&\\3&&\end{matrix}$	$(7,8)\ (8,4)$

8	$\begin{array}{\|c\|c\|c\|c\|} 1 & 2 & 5 & 8 \\ 3 & 7 \\ 6 \end{array}$	$\begin{array}{\|c\|c\|c\|c\|} 1 & 4 & 6 & 7 \\ 2 & 5 \\ 3 \end{array}$	$(8, 4)$
9	$\begin{array}{\|c\|c\|c\|c\|} 1 & 2 & 4 & 8 \\ 3 & 5 \\ 6 & 7 \end{array}$	$\begin{array}{\|c\|c\|c\|c\|} 1 & 4 & 6 & 7 \\ 2 & 5 \\ 3 & 8 \end{array}$	

For the moment, call $p = (p_i)$ an admissible sequence if (1) the p_i are positive integers and (2) all the p_i are distinct. Define $\{p\}$ to be the set of all the p_i. Similarly, if P is a tableau, let $\{P\}$ be the set of its entries.

7.1. Theorem. *The map taking (p, q) to (P, Q) is a bijection between pairs p, q of admissible sequences of the same length and pairs of tableaux P, Q of the same shape with $\{P\} = \{p\}$, $\{Q\} = \{q\}$.*

Proof. How to go backwards from the pair P, Q to the input q, p? This is done in a series of simple steps. At any moment, we have the tail of the array of pairs (q_i, p_i) for $i > k$, and we want to find (q_k, p_k). Then q_k will be the maximum entry in the current Q. It will be at the end of a row of the current Q, say row j. We remove it from that row. To find p_k, we reverse the bumping cascade. That is to say we first remove the item c_j from the end of row j of P. Then while $j > 0$ we scan through row $j - 1$ of P to find last where $P_{j,\ell} < c_j$. Set $c_{j-1} = P_{j,\ell}$, and loop. At the end, $p_k = c_0$. ∎

If we apply this to permutations σ, giving rise to sequences (i, σ_i), this leads to:

7.2. Corollary. *The map $\sigma \mapsto (P_\sigma, Q_\sigma)$ is a bijection of the group of permutations with pairs of Young tableaux of the same shape and entries in $[1, n]$.*

Here is what we get for $\mathfrak{S}_3$:

σ	word expression	P	Q
1 2 3	I	$\begin{array}{\|c\|c\|c\|} 1 & 2 & 3 \end{array}$	$\begin{array}{\|c\|c\|c\|} 1 & 2 & 3 \end{array}$
2 1 3	s_1	$\begin{array}{\|c\|c\|} 1 & 3 \\ 2 \end{array}$	$\begin{array}{\|c\|c\|} 1 & 3 \\ 2 \end{array}$
2 3 1	$s_1 s_2$	$\begin{array}{\|c\|c\|} 1 & 3 \\ 2 \end{array}$	$\begin{array}{\|c\|c\|} 1 & 2 \\ 3 \end{array}$
1 3 2	s_2	$\begin{array}{\|c\|c\|} 1 & 2 \\ 3 \end{array}$	$\begin{array}{\|c\|c\|} 1 & 2 \\ 3 \end{array}$
3 1 2	$s_2 s_1$	$\begin{array}{\|c\|c\|} 1 & 2 \\ 3 \end{array}$	$\begin{array}{\|c\|c\|} 1 & 3 \\ 2 \end{array}$
3 2 1	$s_1 s_2 s_1 = s_2 s_1 s_2$	$\begin{array}{\|c\|} 1 \\ 2 \\ 3 \end{array}$	$\begin{array}{\|c\|} 1 \\ 2 \\ 3 \end{array}$

The **Robinson-Schensted** correspondence is the bijection defined in this section between permutations and pairs (P, Q) of tableaux of the same shape. With the advent of cells and W-graphs (in [Kazhdan-Lusztig:1979]) it has acquired a new significance.

8. Delayed pair processing

In the process described in the previous section, we are given a sequence of pairs (q_i, p_i). In each step, a single integer p_i is inserted into a tableau, and q_i is inserted into a second tableau. Thus tableaux are modified (or enlarged) upon reading each new input pair. But here also one can delay reinsertion. In this way, one constructs only a pair of corresponding rows of P and Q by postponing insertion in lower rows until one has completely constructed the first row.

First of all, I now allow as input any sequence of pairs (p_i, q_i) with q_i not necessarily equal to i, but at least ordered so that $q_i < q_j$ if $i < j$. We can apply the previous process to such a sequence, placing q_i where before we placed i. This generalization will now prove significant.

Instead of inserting a pair (q_i, p_i) and getting two new tableaux, the basic step now will be this:

> **Delayed pair reinsertion.** *Insert p_i into the row P. (a) If p_i is appended to the row, we append q_i to the second row Q. Otherwise (b) p_i will replace some r_ℓ. This does not change the shape of P, and we do not change Q at all. Add the pair (q_i, r_ℓ) at the end of a new input sequence.*

Here is how our example goes:

1			$(1,6)\,(2,3)\,(3,1)\,(4,7)\,(5,2)\,(6,5)\,(7,8)\,(8,4)$
2	$\boxed{6}$	$\boxed{1}$	$(2,3)\,(3,1)\,(4,7)\,(5,2)\,(6,5)\,(7,8)\,(8,4)$
3	$\boxed{3}$	$\boxed{1}$	$(3,1)\,(4,7)\,(5,2)\,(6,5)\,(7,8)\,(8,4)$ $(2,6)$
4	$\boxed{1}$	$\boxed{1}$	$(4,7)\,(5,2)\,(6,5)\,(7,8)\,(8,4)$ $(2,6)\,(3,3)$
5	$\boxed{1}\boxed{7}$	$\boxed{1}\boxed{4}$	$(5,2)\,(6,5)\,(7,8)\,(8,4)$ $(2,6)\,(3,3)$
6	$\boxed{1}\boxed{2}$	$\boxed{1}\boxed{4}$	$(6,5)\,(7,8)\,(8,4)$ $(2,6)\,(3,3)\,(5,7)$
7	$\boxed{1}\boxed{2}\boxed{5}$	$\boxed{1}\boxed{4}\boxed{6}$	$(7,8)\,(8,4)$ $(2,6)\,(3,3)\,(5,7)$
8	$\boxed{1}\boxed{2}\boxed{5}\boxed{8}$	$\boxed{1}\boxed{4}\boxed{6}\boxed{7}$	$(8,4)$ $(2,6)\,(3,3)\,(5,7)$
9	$\boxed{1}\boxed{2}\boxed{4}\boxed{8}$	$\boxed{1}\boxed{4}\boxed{6}\boxed{7}$	$(2,6)\,(3,3)\,(5,7)\,(8,5)$

and now one builds a new row from the input $(2,6)$, $(3,3)$, $(5,7)$, $(8,5)$. This will produce yet another input made up of bounced pairs. And so on.

9. Symmetry

In this section I'll prove Theorem 1.2. I'll start by reintroducing by reintroducing **rank lists**, but in a form now adapted to our use of sequences of pairs (q_i, p_i), ordered according to value of q_i. This now associates to each possible rank the list of all pairs (q_i, p_i) with that rank. Each of these lists is sorted so that new items are put on top of a current list. We have dealt with something like this before, in the course of developing Schensted's row-insertion process, and just as before these rank lists are constructed as we read in the pairs. Here is the new and extended version, with the same input sequence $[6, 3, 1, 7, 2, 5, 8, 4]$:

	$1 \mid 6$	
Rank	0	
Sequence read	6	

	$2 \mid 3$	
	$1 \mid 6$	
Rank	0	
Sequence read	6	3

	3 ¦ 1						
	2 ¦ 3						
	1 ¦ 6						
Rank	0						
Sequence read	6	3	1				

	3 ¦ 1						
	2 ¦ 3						
	1 ¦ 6	4 ¦ 7					
Rank	0	1					
Sequence read	6	3	1	7			

	3 ¦ 1						
	2 ¦ 3	5 ¦ 2					
	1 ¦ 6	4 ¦ 7					
Rank	0	1					
Sequence read	6	3	1	7	2		

	3 ¦ 1						
	2 ¦ 3	5 ¦ 2					
	1 ¦ 6	4 ¦ 7	6 ¦ 5				
Rank	0	1	2				
Sequence read	6	3	1	7	2	5	

	3 ¦ 1						
	2 ¦ 3	5 ¦ 2					
	1 ¦ 6	4 ¦ 7	6 ¦ 5	7 ¦ 8			
Rank	0	1	2	3			
Sequence read	6	3	1	7	2	5	8

	3 ¦ 1						
	2 ¦ 3	5 ¦ 2	8 ¦ 4				
	1 ¦ 6	4 ¦ 7	6 ¦ 5	7 ¦ 8			
Rank	0	1	2	3			
Sequence read	6	3	1	7	2	5	8

This table amounts to a slightly different display of material already examined in §1.

A number of things should be clear from the way the lists are constructed: (a) the process is valid for any sequence of pairs (q_i, p_i) in which all p_i and all q_i are distinct; (b) each rank list is ordered from bottom to top by increasing size of the q_i, and in decreasing size of p_i, *as long as the input sequence is sorted by increasing size of the* q_i.

It is valuable to compare these lists to where we are after we have inserted a sequence into a single row:

1	2	4	8

1	4	6	7

$$(2,6)\ (3,3)\ (5,7)\ (8,5)$$

(a) As we already know, the values of p_i in the final P-row we see are the same as the values of p_i in the array of *top* pairs in the rank lists. (b) But also, the values of q_i in the final Q-row we see are the same as the values of q_i in the array of *bottom* pairs in the rank lists. This is because these values are set when a rank is first occupied. Thus we can read off fromhe rank lists the pair of rows $P = (1, 2, 4, 8)$ and $Q = (1, 4, 6, 7)$.

But in addition there is one significant feature that is a bit more subtle. If we look at the final rank lists:

	$3 \mid 1$						
	$2 \mid 3$	$5 \mid 2$	$8 \mid 4$				
	$1 \mid 6$	$4 \mid 7$	$6 \mid 5$	$7 \mid 8$			
Rank	0	1	2	3			
Sequence read	6	3	1	7	2	5	8

we see from the recipe for delayed pair reinsertion that the input sequence for the next row of the P-tableau is made up of shifted values of q and p in the rank lists. To be precise, let $r_{n,i} = (q_{n,i}, p_{n,i})$ be the list of pairs of rank n. Thus in the example above, $r_{0,0} = (1, 6)$, $r_{0,1} = (2, 3)$. To get the next input, we scan through all the rank lists starting at rank 0, taking the p values from the pairs $r_{n,i}$ and the q-values from $r_{n,i+1}$. Thus in the example above we have

$$(2,6)(3,3)(5,7)(8,5) = (q_{0,1}, p_{0,0})(q_{0,2}, p_{0,1})(q_{1,1}, p_{1,0})(q_{2,1}, p_{2,0})\,.$$

This is always true: *the next input sequence is made up of pairs p from the $r_{n,i}$ and q from $r_{n,i+1}$ (sorted according to size of the values of q)*. In other words, the rank lists encode very neatly the results of insertion into a single row. This concludes the proof of Theorem 1.2.

Keep in mind that the *content* of the rank lists is a canonical invariant of the original set of pairs (q_i, p_i), independent of any ordering assigned to them. This is because the rank is that in a partially ordered set defined by the pairs.

10. References

1. Donald Ervin Knuth, 'Permutations, matrices, and generalized Young tableaux', *Pacific Journal of Mathematics* **34** (1970), 709–727. Also in the collection **Selected papers on discrete mathematics**, CSLI, Stanford, 2003.

2. ——, **Sorting and Searching**, volume III of **The Art of Computer Programming**, Addison-Wesley, 1975.

3. C. Schensted, 'Longest increasing and decreasing subsequences', *Canadian Journal of Mathematics* **12** (1963), 117–128.