Plactic key agreement

Daniel R. L. Brown

May 12, 2021

Abstract

Plactic key agreement is a new key agreement scheme that uses Knuth's multiplication of semistandard tableaus from combinatorial algebra. The security of plactic key agreement relies on the difficulty of some computational problems, such as division of semistandard tableaus.

Division by erosion uses backtracking to divide tableaus. Division by erosion is estimated to be infeasible against public keys of 768 or more bytes. If division by erosion is the best attack against plactic key agreement, then secure plactic key agreement could be practical.

1 Introduction

Knuth's [Knu70] multiplication of semistandard tableaus is described in §2. (Some history of semistandard tableaus is covered in §A.1.)

Plactic key agreement uses multiplication of semistandard tableaus. Alice and Charlie agree on a secret key as follows. All keys are semistandard tableaus. Alice generates her private key a. Charlie generates his private key c. Alice and Charlie both have the same (public) base key b. Alice computes her public key $d = ab$ and delivers d to Charlie. Charlie computes and delivers his public key $e = bc$ to Alice. Alice computes a secret key $f = ae$ from her private key a and Charlie's public key e. Charlie computes a secret key $g = dc$.

The secret keys f and g agree, because multiplication of semistandard tableaus is associative and $f = a(bc) = (ab)c = g$.

Plactic key agreement can be considered an instance of Berenstein and Chernyak's modification [BC04] of Diffie and Hellman's exponential key agreement [DH76], by setting the Bernstein and Chernyak's semigroup[1] to be the plactic monoid[2]: which is the set of all semistandard tableaus, with Knuth's multiplication as the associative binary operation.

Division of semistandard tableaus is discussed in §3. Division can be used as an attack against plactic key agreement as discussed in §4.2. The secret key f can be computed from the (public) keys d, b, e using the formula $f = (d/b)e$. Therefore, the security of plactic key agreement relies on the difficulty of dividing semistandard tableaus.

Division by erosion, described in §3.3, is a method to divide semistandard tableaus. It computes d/b by deleting each entry of b from d, in the reverse order that the multiplication algorithm inserted entries of b into d. A catch is that multiplication is non-cancellative. At each stage, there can be multiple choices of which entries of d to delete. Some deletion choices can turn out to be incompatible with the overall division task. Division by erosion uses backtracking to correct these incorrect deletion choices. Empirically, division by erosion seems slow, taking an average of about $2^{0.3m}$ attempted deletions for m-entry tableau b (with entries in a set of 64 values). Extrapolating this empirical evidence to $m = 512$, takes this to 2^{153} deletion attempts, which should be infeasible.

Other possible division algorithms include: division by trial multiplication (§3.4), which searches through the key space of a, and division by max-algebra matrices (§3.5), which uses Johnson–Kambites [JK19] tropical matrix representations of the plactic monoid. These other division algorithms are predicted to be slower than division by erosion, if a and b belong to suitably large key spaces.

If division by erosion is the fastest attack against plactic key agreement, then secure plactic key agreement could be practical.

The security analysis of plactic key agreement is in its very early stages.

2 Knuth multiplication of tableaus

This section describes Knuth multiplication of semistandard tableaus [Knu70].

[1] A semigroup is a set with an associative binary operation.

[2] A monoid is a semigroup with an identity element.

2.1 Semistandard tableaus

A (**semistandard**) **tableau** is a two-dimensional grid-aligned arrangement of entries, meeting the following requirements.

1. The sequence of rows are
   (a) left-aligned,
   (b) sorted by row length (shortest row at the top, longest row at the bottom, repeated lengths allowed)
2. The entries of each row are
   (a) sorted (lowest at left, highest at right, repeated entries allowed).
3. The entries of each column are:
   (a) sorted (highest at top, lowest at bottom),
   (b) all distinct (no repeated entries allowed).

Two examples of semistandard tableaus with single-digit entries are:

$$a = \begin{array}{l} 4 \\ 334 \\ 1233 \end{array} \qquad\qquad b = \begin{array}{l} 2 \\ 13 \end{array} \tag{1}$$

Note the following special considerations. An empty semistandard tableau is possible, with no entries, and no rows. Many authors, including Knuth [Knu70], reverse the order of rows. (Each column would have lowest entries at the top.) It is sometimes convenient to consider there to be extra empty rows above the top row.

Some mnemonics: rows may repeat but columns cannot, the entry orderings follow the Cartesian coordinate orderings, like this:

$$\begin{array}{c|l} & 4 \\ \vee & 334 \\ & 1233 \\ \hline & \leq \end{array} \qquad\qquad \begin{array}{ccccccc} 4 & & & & & & \\ \vee & & & & & & \\ 3 & \leq & 3 & \leq & 4 & & \\ \vee & & \vee & & \vee & & \\ 1 & \leq & 2 & \leq & 3 & \leq & 3 \end{array} \tag{2}$$

2.2 Overtableaus and row reading

This section defines a few basic tableau operations, which will help define Knuth multiplication.

The **size** $|t|$ of the tableau t is the number of entries in the tableau. For example, the tableaus a and b from (1), have sizes:

$$|a| = 8 \qquad\qquad |b| = 3 \tag{3}$$

The empty tableau has size 0.

The **bottom row** $\underline{t}$ of a tableau t is the last row. The bottom row contains the least entry of each column. The **overtableau** $\overline{t}$ is the tableau that sits above the bottom row $\underline{t}$. In other words, the overtableau $\overline{t}$ is just t with bottom row $\underline{t}$ removed. For the example tableaus a and b from (1), we have:

$$\overline{a} = \begin{array}{l} 4 \\ 334 \end{array} \qquad\qquad \overline{b} = \begin{array}{l} 2 \end{array} \tag{4}$$

$$\underline{a} = \begin{array}{l} 1233 \end{array} \qquad\qquad \underline{b} = \begin{array}{l} 13 \end{array} \tag{5}$$

Some special cases: if t has only one row, then its overtableau is the empty tableau; if t is empty, then both the overtableau and the bottom row are empty.

The **row reading** $\rho(t)$ of a tableau t is the concatenation of all the rows of t, starting with top row at the left, and ending the bottom row at the right. For the example tableaus a and b from (1), the row readings are:

$$\rho(a) = \begin{array}{l} 43341233 \end{array} \qquad\qquad \rho(b) = \begin{array}{l} 213 \end{array} \tag{6}$$

When we need to refer to individual entries of a tableau, we can write t_i for the i^{th} entry of $\rho(t)$, starting with $i = 1$ at the left. So,

$$\rho(t) = t_1 t_2 \dots t_{|t|}. \tag{7}$$

Note that we sometimes write t_i to indicate multiple different tableaus: if the context is not enough to distinguish whether t_i means an entry of t, or a distinct tableau, then write $\rho(t)_i$ for the i^{th} entry of the row reading.

2.3 Multiplication

To multiply tableaus t and u, compute $v = tu$ as follows:

1. Initialize $v = t$.

2. For $k = 1$ to $k = |u|$, insert entry $x = u_k$ into tableau v, as described below.

3. To **insert** entry x into tableau v, do one of the two actions below (whichever is required by the stated conditions):

   (a) If appending entry x to bottom row $\underline{v}$, written $\underline{v}x$, results in a sorted row (lowest to highest), then

      i. replace bottom row $\underline{v}$ by $\underline{v}x$

   (b) Else (if appending x to $\underline{v}$ does not result in a sorted row),

      i. Let r be the least entry in bottom row $\underline{v}$ that is larger than x,

      ii. Replace the leftmost copy of r by x, modifying $\underline{v}$.

      iii. Insert r into overtableau $\overline{v}$ (so, entry x "bumps" entry r up into the rows above).

Note that multiplication adds sizes ($|tu| = |t| + |u|$), because each inserted entry increases the size of v by 1. Insertion has been defined recursively, since the insertion procedure uses the insertion, but different inputs (the replace entry and the overtableau). Insertion could have instead been defined iteratively, as loop starting from the bottom row, bumping up to the next row, until appending to the end of a row is possible.

2.4 Example of Knuth multiplication

As an example, let us multiply $t = a$ and $u = b$, from (1). We let $v = a$, as the initialization. Since $\rho(b) = 213$, we must first insert $x = b_1 = 2$ into v.

The bottom row $\underline{v}$ is now 1233. The concatenation $\underline{v}x = 12332$ is not sorted lowest to highest because the last two entries have $3 > 2$. So, we must apply the second option for insertion, which bumps up an entry of the bottom row. The least entry r of $\underline{v}$ that is larger than $x = 2$ is $r = 3$. So, we replace the bottom row 1233 by 1223. The replaced symbol $r = 3$ is bumped up into the overtableau $\overline{v}$. To do the bumping, we first look at the the overtableau $w = \overline{v}$, which is

$$w = \overline{v} = \begin{array}{l} 4 \\ 334 \end{array} \tag{8}$$

Because we are now trying to insert 3 into $\overline{v}$, let us now write $x = 3$. We see that $x = 3$ cannot be appended to the bottom row $\underline{w} = 334$ of w, because 3343 is not sorted. Another entry must be bumped up. In this case, the smallest entry larger than $x = 3$ is $r = 4$, so we must bump up 4. The bumped-up entry, 4, can be appended to the very top row, giving a new top row, 44. So, after insertion of 2 into v, we get a new value for tableau v:

$$v = \begin{array}{l} 4\mathbf{4} \\ 33\mathbf{3} \\ 12\mathbf{2}3 \end{array} \tag{9}$$

where the changed entries have been shown in bold.

The next entry of b to insert is $b_2 = 1$, which must be inserted into this new v. From our experience of the previous insertion, we can see what happens a little more quickly: 1 bumps 2 from the first row (first from the bottom), which bumps 3 from the second row, which bumps 4 from the third row. This time, the last bumped entry 4 gets appended to the empty row above, creating a new non-empty row. The new v is

$$v = \begin{array}{l} \mathbf{4} \\ \mathbf{3}4 \\ \mathbf{2}33 \\ 1\mathbf{1}23 \end{array} \tag{10}$$

where, the changed entries of the new v have (again) been shown in bold. Note that the changed entry in the bottom row is the inserted entry b_2, while the changed entries in overtableau were the bumped entries.

The final entry of b to insert is $b_3 = 3$. This entry gets appended to the bottom row (because appending it gives a sorted row), which gives the new v, and also the final multiplication of a and b,

$$ab = v = \begin{array}{l} 4 \\ 34 \\ 233 \\ 1123\mathbf{3} \end{array} \tag{11}$$

with the single changed entry shown in bold.

2.5 The plactic monoid

Recall that a **monoid** is a set with an associative binary operation, with an identity element. Often, the operation is written as multiplication, in which

case, the associative law means that $a(bc) = (ab)c$ for all a, b, c. If clear from context, the identity element is written as 1, in which case, the identity element condition means that $1a = a = a1$ for all a.

Knuth's multiplication of semistandard tableaus is an associative binary operation. Associativity is not obvious: Knuth's realization that semistandard tableaus could be multiplied associatively was a major insight.

The identity element for Knuth's multiplication of semistandard tableaus is the empty tableau. For tableaus, the notation 1 for the identity element conflicts with tableau of single entry of value 1, so we can instead write ϵ. Then $\epsilon t = t = t\epsilon$ for any semistandard tableau t.

Therefore, the set of semistandard tableaus with Knuth multiplication meets the defining axioms of a monoid. This monoid is now called the **plactic monoid** [LS81].

The plactic monoid is **non-commutative**: there are tableaus a and b such that $ab \neq ba$ (and non-commuting is typical).

The plactic monoid is **non-cancellative**: there are distinct tableaus a, e, b with $ab = eb$ (for any a, b, it is typical for there to exist such an e).

An alternative definition of the plactic monoid is to use generators and relations to give a **monoid presentation**, which is summarized below. The monoid generators are possible entry values, which is some ordered set. The relations are the **Knuth relations**, which say that $yxz = yzx$ if $x < y \leq z$ and $xzy = zxy$ if $x \leq y < z$. Knuth [Knu70] proved each congruence class of words under these relations has a unique representative that is the row reading of a semistandard tableau.

This alternative definition allows us to think of each word as being some alternative representation of a semistandard tableau.

2.6 The Robinson–Schensted algorithm

Knuth multiplication is almost equivalent to the Robinson–Schensted algorithm [Rob38, Sch61].

The Robinson–Schensted algorithms takes any word $u = u_1 u_2 \dots u_n$ (not necessarily a row reading of a tableau), and then inserts the entries u_i (in order) into a tableau, starting from an empty tableau, using the same insertion method as in Knuth multiplication. The result is always a semistandard tableau, which is traditionally written $P(u)$.

In the monoid presentation of the plactic monoid, a word w represents a tableau t if and only if $P(w) = t$.

Knuth multiplication has an alternative definition in terms of the Robinson–Schensted algorithm as:

$$ab = P(\rho(a)\rho(b)). \tag{12}$$

Consequently, for any semistandard tableau t, we have $t = P(\rho(t))$.

The **column reading** $\gamma(t)$ of tableau t is a word formed by reading the columns from left to right, while reading each individual column from top to bottom. In other words, separate the columns, topple each column to the left, and concatenate the fallen columns. Perhaps surprisingly, $P(\gamma(t)) = t$.

The Robinson–Schensted can also be interpreted as the Knuth multiplication of n single-entry tableaus.

2.7 A C program for Knuth multiplication

Table 1 is a C program that can run Knuth multiplication of semistandard tableaus (and the Robinson–Schensted algorithm).

```
#include <stdio.h>  // gcc knuth_mult.c -o knuth_mult
#define T(a,b)(a^=b,b^=a,a^=b,1)
enum{L=1000000};typedef char*s,S[L+1];typedef int i;typedef void _;
i knuth(i i,s w){return(w[2]<w[i-1])&(w[0]<=w[i])&&T(w[1],w[(i+1)%3]);}
_ robinson(s w,i j){i i;  for(;j>=2&&w[j]<w[j-1];j--)
    for(i=1;i<=2;i++)  for(;j>=2&&knuth(i,w+j-2);j--) ; }
i main(_){S W={};s w=W;i i=0,o=0,c;
  while(i<L&&(c=getchar())!=EOF)if(c&&'\n'!=c)w[i]=c,robinson(w,i++);
  for(;(o>*w?puts(""):1)&&*w;w++)putchar(o=*w);}
```

Table 1: A C program for Knuth multiplication

The simplicity of Knuth multiplication can be measured by the brevity of the C program in Table 1.

Note that the C program stores each entry with the C `char` data type (almost always a byte). On many computer systems, the standard input and standard output of `char` values uses the ASCII encoding of characters. In this case, the C program allows tableau entries to be single digits, or letters

(latin alphabet), or even punctuation, with the entry-sorting based on the (somewhat arbitrary) ASCII ordering.

```
 !#"$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~
```

Note that the C program uses the Knuth relations to insert entries into the row reading representation of tableaus.

Table 2 shows an example of a shell[3] session using the C program for multiplication.

```
$ ./knuth_mult <<<'
> 4
> 334
> 1233
> ''
> 2
> 13
> '
4
34
233
11233
```

Table 2: A shell session using of the C program for multiplication

3 Division of semistandard tableaus

A binary operator / is a (**right**) **division operator** if

$$((ab)/b)b = ab \tag{13}$$

for all a, b. Call / a **divider** (for brevity).

Note that other definitions for a divider are possible. In other areas of algebra, division is often required to **cancel** multiplication, meaning that

[3]This session uses the "bash" shell, and might not work with other shells.

$(ab)/b = a$ for all a, b. A cancelling divider is not possible Knuth multiplication, because the plactic monoid is non-cancellative. The less strict requirement (13) for division is relevant to the security of plactic key agreement, so that is the definition that we will use.

3.1 A division challenge

Independent efforts to devise algorithms for division of semistandard tableaus can potentially result in faster algorithms. Some researchers might achieve more independent thought by not learning existing division algorithms. The C program for multiplication can be used to generate examples on which to try potential division algorithms. Some researchers might be more motivated by being given challenges.

For those researchers, a challenge division problem is given in Table 3 and Table 4, generated as follows.

Random words a and b of length 512 were generated, drawn from a set of 64 possible characters (from the standard `base64` binary-to-text encoding system). The value of b is shown in Table 3 (broken into several lines, for convenience). The Robinson–Schensted can put b into its unique semistandard tableau (but this is not shown, both to save space and to better show the randomness of b). The value of a is not included, because it would lead to a solution of the challenge.

```
gnxkOR2uN/j/sWwxNHcMKh1DaMaV4ifNUkJhjr9WWVAHVA5FNRdNYt1/bNGYuq5lZIjIiGtxjdVl
T+shNNf5NnYWawpQPJZStxH376j3JQgqwLLy0o5dq4vLeUJrSyoNUGfFB+dQawvYYRTQH+tJZQiA
usuD+VTNYkqBoVnl0Vt2CDKGNhNCdiYzf7O6IhgMJVmQxgmAGUPQwOinni6asO+sqodfogSB4BOD
g3UTUl5xnDOALslyiSm3A7vO+8kOr8976RCTf19I+ZGWfihspGQUdcCwrcCmYRow31AEWMwbKnPL
D41maUNHsOBVtJJU58RZcjubTZqnga1f13Bsb/lLn0rXg73vDhCEpbr+yUi6ZOYc+mZW+hB2Cvih
6vJ3km3wxaMag86a2i+k1r9dOmcKTITJwjONvr1mDlpISCsmMwTbMcE7ddvbVjGw+TpP9xqQPaOT
BMRkW2zP8zcc+8kAr1XClSeb3LKBriZYkHY8OP7zaDJh3JJNNgwY/lpf
```

Table 3: Value b (as generated) of the challenge division problem

The tableaus a and b were multiplied to get $d = ab$, which is shown in Table 4, in semistandard tableau form.

The challenge is to compute d/b. Recall that this means finding a tableau u such that $ub = d$. A solution exists, namely $u = a$, but there are likely many other solutions, with distinct semistandard tableaus, because Knuth multiplication is non-cancellative.

```
x
v
u
p
ny
mwz
ktxy
jsux
ipsux
gopst
ekorrxz
cjloqvy
bhkmntx
aeikmqwy
Ybgilpvxyz
VZdhkouvvxz
UXbfhnqstvxz
TWaeglprssty
SVZbfhmoorsxxxz
RRVZacjmmnruuww
PPTXZbejkmnttuv
OOQVXabdjllmnquvw
MNPTUXYchkkllltuvx
LMOOSVXZbgijjkqqtvwx
KLMNOSWYYbhiiioopuvw
JKKMMRUWXabffghjltuux
IJJKLPTUWZacdefiiiksv
HHIJKOQRUWXXbbccghjrtvwx
GGHHILLPSSUVVWZbegijosuwy
FFGGHIKNOQTUUVXaaegiinqtxxx
DEEFGGJMMOOSSTWYYbeggmnouuux
CDDDFFGJLLMNRRSTWWZffgiknstvvwy
ABBCDEFHIKLMNOQRUVYbbdhijooootuz
9AABBCEFFHJKLNOQTUVZaadfiklllmpqswx
89999BCDDDHHKKLMNSTTTYaacchhhjkoqtv
788889BCCCFGIIJKKPRRRUUWZZbbffgjnooqqy
67777888ABCFFFGHHMNNNQSVYYYabdfhikloooorx
566667779999ABEFGGJJKLMPPQRUUYaddggiimnnoswwwy
44555666678889BBDDDGHILLMOOQRSTUVXYdddllnnqrrss
3333344455778888BBEGGGIKLNNNNNQUVVVZccfhikmmmrsww
222222333446777 79AABBDDGJJJJJKLMNTTUUWdffggijknpuw
1111112222223445599AAAAABCCCDGHIJPQSTUUWYZbcghilmmpvvw
000000011111122333355578 8ABBCCDEIJLMNNQQRRabbbccdhhijkmrrsv
//////////0000001113333566667 89ACGJJJKMMOOSTTVZZZaaaaglmpqrrvvwwxz
++++++++++++++++++++++++/11333788BBBCDHJJJJJNNNNNOOOOOPPPTTWXYYYbcefkklpwzz
```

Table 4: Value d (in tableau form) of the challenge division problem

The division algorithms described in this report are estimated to take an infeasible amount of time to compute d/b. Therefore, if a researcher solves this challenge, then a faster division algorithm seems to have been found.

3.2 Left division reducible to right division

The plactic monoid is **anti-automorphic**, meaning that it is isomorphic to its mirror image, where one swaps the left and right tableaus in the definition of multiplication. (The anti-automorphism swaps values of high and low entries, and also reverses the order of the row readings.)

The anti-automorphism means that left division, an operator $\backslash$ such that $b(b \backslash (ba)) = ba$, can be reduced to right division.

3.3 Division by erosion

This section (§3.3) explains **division by erosion**, which is an algorithm to divide semistandard tableaus, set out on this report.

3.3.1 An overview

Eroding b from $d = ab$ means trying to delete the entries of b from d, one-at-a-time, in the opposite order from which they were inserted. Each deletion bumps entries down the tableau, starting from a peak in the tableau (as defined in §3.3.2). Erosion tries each peak of d to see if it deletes the last entry x of b that needs to be deleted. On a match, it recursively uses erosion to delete more entries of b from d. Whenever a match fails, erosion backtracks the attempted deletion, and tries to delete and recursively erode from a new peak of d.

Note the following metaphoric terminology. The term *plactic* is related to plate tectonics, multiplication of tableaus is a little like two mountains colliding and merging into one larger mountain. By contrast, *erosion* amounts to the opposite process, the larger mountain eroding down into a smaller mountain. More specifically, the erosion algorithm tries to remove random small pieces of the larger mountain, pushing down from various points at the upper surface of the mountain. The deletion down the rows can be considered as a stream, eroding the mountain.

3.3.2 Peaks of a tableau

A **peak** p of a tableau is an entry p with no entry above it and no entry to the right. Equivalently, a peak is an entry that is both the top of a column and the end of a row. In other words, a peak is an upper-right corner of a tableau. For example, tableau $d = ab$ from (11) has four peaks, which are are shown in bold below:

$$\begin{array}{l} \mathbf{4} \\ 3\mathbf{4} \\ 23\mathbf{3} \\ 1123\mathbf{3} \end{array} \tag{14}$$

When inserting a new entry x into a tableau v to get a larger tableau w, the last entry of w to be changed will be a peak of w. This new peak is the only entry of w in a new position that was not occupied in v. In other words, insertion always ends at a (new) peak. This view suggests that peaks should be the starting point for process opposite to insertion (deletion, defined in §3.3.3).

Note that different peaks can have the same value, so **peak** includes the position of entry, not just its value.

3.3.3 Deletion

Deletion, defined below, starts at a given peak p of a given tableau t, and ends by producing a smaller tableau s (a modification of t) and results in an entry x being deleted from the bottom row of t. In other words, the input is tableau t and the peak p. The tableau s and the deleted entry x are the output.

1. Initialize s to be the tableau t.
2. Initialize x to be value of entry p.
3. Let r be positive integer such that p is in the row r of s (the bottom row is row 1, and then we count up).
4. Remove p from (the end of) row r of s (so now $|s| = |t| - 1$).
5. While $r > 1$, repeat the following
   (a) Reduce r by 1.

(b) In row r of s, let y be the largest entry with $y < x$.

(c) In row r of s, replace the rightmost copy of y by x.

(d) Update x to value y.

6. Output the tableau s and the final entry value x.

Note that deletion is the opposite of insertion, in the sense that inserting entry x into s results in t. Knuth [Knu70] defined deletion.

3.3.4 Examples of deletion

Each of the four peaks in (14) results in its own deletion which are illustrated below:

$$\begin{array}{cccc} * & 4 & 4 & 4 \\ \mathbf{4}4 & 3* & 34 & 34 \\ \mathbf{3}33 & 23\mathbf{4} & 23* & 233 \\ 1\mathbf{2}233 & 11\mathbf{3}33 & 11\mathbf{3}33 & 1123* \\ \hline 1 & 2 & 2 & 3 \end{array} \tag{15}$$

The new smaller tableau s is shown above the line, and the deleted entry x is shown below the line. The entries of s that underwent replacement are shown in bold. The former position of the peak of t is shown as $*$. The peak $*$ is not entry of s.

Note that sliding the bold characters up a row shows the process of re-insertion of the deleted entry. Referring to erosion metaphor, the bold characters indicate the path of a stream in the erosion of the mountain.

3.3.5 Erosion of a tableau by a word

Erosion takes as input a tableau v and a word $b = b_1 \dots b_m$. It either fails, or returns as output a smaller tableau t.

Erosion is a recursive procedure. The erosion v by b, will require the erosion of smaller tableaus by shorter words. The number of sub-erosions needed can be large, and the exact number seems hard to predict. It is easier to describe erosion as a recursive procedure rather than an iterative procedure.

To erode tableau v by word $b_1 \dots b_m$, do the following

1. If $m = 0$, then stop and output $t = v$ (successfully).

2. If $m > 0$, then for each peak p of v do the following:

   (a) Apply deletion to v, starting from p, getting a smaller tableau s and a deleted entry x.

   (b) If $x \neq b_m$, then continue (to the next iteration of the loop over the peaks of v, if any peaks are left).

   (c) If $x = b_m$, then (recursively) run erosion on tableau s by word $c = b_1 \dots b_{m-1}$, and do the following depending on the result:

      i. If erosion of s by c succeeds, with result of tableau q, then let $t = q$ and stop (so that $t = q$ is the output of eroding v by b).

      ii. If erosion of s by c fails, then continue (to the next iteration of the loop over the peaks of t, if any peaks are left).

3. At this point, the loop over all the peaks p of t has finished, with none of the peaks successfully leading to an output tableau. In this case, stop, and indicate the failure of erosion of v by b.

3.3.6 Division by erosion

Division by erosion means to divide tableau d by tableau b by applying erosion of tableau d by word $\rho(b)$ (the row reading of b).

More generally, one can use erosion of d by any word w representing b, meaning that $P(w) = b$.

3.3.7 Example of erosion

We now divide tableau $d = ab$ from equation (11) by b from equation (1). Since $\rho(b) = 213$, we see that we need to delete $m = 3$ entries. The first entry to delete is $b_3 = 3$. The four peaks of d were listed already in (14), and the results of deletions from these four peaks were listed already in (15). Only one of the four peaks results in the deletion of $b_3 = 3$, which happens to be the lowest peak (the rightmost peak).

The deletion of 3 gives us a smaller tableau s that also has four peaks.

The deletions of the four peaks of s are as follows:

$$\begin{array}{cccc} * & 4 & 4 & 4 \\ \mathbf{4}4 & 3* & 34 & 34 \\ \mathbf{3}33 & 23\mathbf{4} & 23* & 233 \\ 1\mathbf{2}23 & 11\mathbf{3}3 & 11\mathbf{3}3 & 112* \\ \hline 1 & 2 & 2 & 3 \end{array} \tag{16}$$

The next entry to delete is $b_2 = 1$, but the only peak of s that results in deleting 1 is the highest peak (the leftmost peak). So, now we continue the erosion process from the leftmost tableau of the four choices above.

The new smaller tableau s has 3 peaks, that get deleted as follows:

$$\begin{array}{ccc} 4* & 44 & 44 \\ 33\mathbf{4} & 33* & 333 \\ 12\mathbf{3}3 & 12\mathbf{3}3 & 122* \\ \hline 2 & 2 & 3 \end{array} \tag{17}$$

The next entry of b to delete is $b_1 = 2$. We see that deletion of two of peaks result in the deletion of 2. Erosion will output of one of these results, depending on the ordering of the peaks in the iteration loop over the peaks.

Deleting from highest peak of s gives back the original tableau a from (1). But suppose that we somehow used a different ordering of the peaks, so that we end up deleting the second highest peak from s. Then division by erosion would give a final answer of

$$d/b = \begin{array}{c} 44 \\ 33 \\ 1233 \end{array} \tag{18}$$

3.3.8 A C program for division by erosion

Table 5 shows a C program that runs one possible version of division by erosion.

An example shell session of running the C program for division by erosion is shown in Table 6.

Note that the interface to the division C program differs slightly from the multiplication C program. The second input b to division is supplied as a command-line argument rather than as part of standard input. In Table 6, the line with two quotes includes a space to cause b to be supplied as a

```
#include <stdio.h> // gcc -O3 erode.c -o divide
#define T(a,b)(a^=b,b^=a,a^=b)
enum{E=20,N=1<<E,L=N/E,A=253};
typedef char *s,S[L+1],**t,*T[A]; char D[N];
typedef int i; typedef void _;
_ insert(t d,i*r,char c){i i=0,j;
  for(; i<A && r[i] && d[i][r[i]-1]>c; i++){
    for(j=0; j<r[i] && d[i][j]<=c; j++);
    T(c,d[i][j]);}
  d[i][r[i]++]=c;}
i delete(t d,i*r,i p){i i,j,q;char c=0;
  for(i=0,q=-1;q<p;i++)q+=(r[i]>r[i+1]);
  for(i--,T(c,d[i][r[i]-1]),r[i]--;i--;T(c,d[i][j]))
    for(j=r[i]-1;j>=0&&d[i][j]>=c;j--);
  return c;}
i erode (t d,i*r,s b,i l){i c,i,p;
  if(0==l--) return 1;
  if('\n'==b[l]) return erode(d,r,b,l);
  for(p=i=0;i<A;i++)p+=(r[i]>r[i+1]);
  for(c=0; p-- && c<=b[l]; insert(d,r,c))
    if(b[l]==(c=delete(d,r,p)) && erode(d,r,b,l)) return 1;
  return 0;}
i slen(char*b){i l=0;while(*b++)l++;return l;}
i divide(s w,s b){i i,r[A]={},j=slen(w);T d={D};
  for(i=1;i<A;i++) d[i]=d[i-1]+1+j/i;
  for(i=0;i<j;i++) insert(d,r,w[i]);
  if (erode(d,r,b,slen(b))){
    for(i=A-1;i>=0;i--) for(j=0;j<r[i];j++) *w++ = d[i][j];
    return *w=0, 1;}
  return 0;}
i main(i n,t b){S W;s w=W;i i=0,o=0,c;
  if(2!=n) return 2;
  while(i<L && (c=getchar())!=EOF)if(c&&'\n'!=c)w[i++]=c;
  if (divide(w,b[1])){
    for(;(o>*w?puts(""):1)&&*w;w++)putchar(o=*w);
    return 0;}
  return 1;}
```

Table 5: A C program for division by erosion

```
$ ./divide <<<'
> 4
> 34
> 233
> 11233
> ' '
> 2
> 13
> '
4
334
1233
```

Table 6: Shell session run of C program for division by erosion

command-line argument, while the line with two quotes in Table 6 does not include a space, to ensure that b is supplied as part of standard input.

3.3.9 Empirical run-times of erosion

Average empirical run-times for eroding d/b, where $d = ab$, with a and b of length $L \leq 60$, drawn uniformly from a generator set of 64 symbols, seem to take about $2^{0.3L}$ deletion attempts.

3.3.10 Variants of erosion

Variants of erosion are possible by choosing different methods of looping through the peaks. Highest-to-lowest and lowest-to-highest are two simple methods. Choosing the peaks in a random order might also help in some cases (perhaps a and b have the property such that the fixed orderings of peaks are slower than random peak orderings).

Shortcuts are also possible. The highest peaks deleted the lowest value of elements. A peak also deletes an element of the bottom in the same column, or in a column further to the right. These observations provide a shortcut, to reduce the amount of tests on non-matching peaks. If there are very many peaks, then perhaps a binary search will be a faster way to find the matching peaks.

Shortcuts to delete only matching peaks could improve the speed of erosion by the inverse of the proportion of matching peaks to non-matching peaks. This is easily upper bounded by approximately $\sqrt{|d|}$, the maximum number of peaks.

Instead of pushing down from the peaks, one can instead try to pull down from the bottom row. Implementations of this worked, but were much slower than erosion. The problem may have been that are multiple choices of which entry to pull down from the row above. Thus deleting a single entry requires exploring many pull downs, backtracking from incompatible pull downs.

The process of deletion is actually part of an enhanced version of the Robinson–Schensted algorithm. On input of a word w, the enhanced Robinson–Schensted outputs a pair $[P(w), Q(w)]$ of semistandard tableaus. The extra tableau $Q(w)$ has the same shape as P (meaning the each row r of Q has the same length as row r of P). The set of entries of Q is exactly the set $1, \ldots, |Q(w)|$. Such a tableau is called a **standard** tableau. Basically, Q records the positions of the peaks when generating $P(w)$ from Q. The peaks are recorded in Q, so deletion can be applied to recover w exactly from $[P(w), Q(w)]$. The enhanced Robinson–Schensted algorithm is a bijection between words and pairs of semistandard tableaus of the same shape, with the second tableau being a standard tableau.

Erosion of tableau t by word b can be interpreted as a simplification of the following algorithm. Loop over standard tableaus s of the same shape as t. Apply the enhanced Robinson–Schensted algorithm to get a word w from $[t, s]$. If $w = eb$ for some word e, then output $P(e)$ as the value of d/b. Erosion simplifies this procedure by not , but only the portion of the standard tableau s needed to check it will match with b.

3.3.11 Estimates for the challenge

For the challenge division problem, we can count the number of standard tableaus s with the same shape as d, using the hook-length formula. This is approximately 2^{4309}. Division by erosion will use at most that many steps, but likely far fewer steps.

Empirical estimates suggest the main influence on the cost of division by erosion is the size of tableau b, with much less influence from the size of a or the size of the alphabet, provided both are not too small. The number of deletions used seem to be exponential in the length in the size of 4b, $at approximately 2^{0.3L}$ deletions if

3.4 Division by trial multiplication

Division by **trial multiplication** means computing d/b as follows. Use a quick method to generate a tableaus from long search list of tableausTableau a_i in the search list does not More precisely, a division algorithm uses trial multiplication if the vast majority of its run-time computation is spent on tableau multiplications $d_i = a_i b$. (So, a division algorithm that spends more time generating the a_i than testing them is not to be considered trial multiplication. In particular, division by erosion, which spends a long time to generate a one-tableau list $[a_1]$, is not to be considered as a trial multiplication.)

Trial multiplication is **strict** if the search list $[a_1, \ldots, a_L]$ does not depend on d. More precisely, strict trial multiplication can generate $[a_1, \ldots, a_L]$ based on *a priori* given infomration about the method used to generate a, such as Alice's random variable she uses to generate a – but strict trial multiplication cannot depend on any information specifc to the target instance of the generated a, which includes $d = ab$.

For small sized a, or a generated from a small set of large tableuas, trial multiplication can be faster than erosion.

The rest of this section discusses some details of trial multiplication, such as how to generate the search list $[a_1, a_2, \ldots, a_L]$.

3.4.1 Guaranteed success

Division by trial multiplication is guaranteed to succeed if it can be guaranteed that the search list contains the a using to generate the input $d = ab$ defining the instance of the division problem.

We assume that the search list is guaranteed to contain a. Given alist $[a_1, \ldots, a_L]$, for which this is not guaranteed, it is usual straightforward to expand the list to ensure it covers all the a can be used to generate the problem instance.

3.4.2 Generating the list as words instead of tableaus

It might be faster to generate the tableaus a_i from a list words $[w_1, \ldots, w_L]$, with $a_i = P(w_i)$, where P is the Robinson–Schensted algorithm that converts words to semistandard tableaus.

3.4.3 Content of a tableau

The (**multiplicity**) **content** $\mu(t)$ of a tableau t is an array $[\mu(t)_x]_x$ of integers such that t contains $\mu(t)_x$ entries with value x. When assuming that all entries of t are taken as positive integers, we write $\mu(t) = [\mu(t)_1, \mu(t)_2, \dots]$.

For the example tableaus a and b from (1), the contents are:

$$\mu(a) = [1, 2, 4, 2, 0, 0, \dots] \qquad \mu(b) = [1, 2, 3, 0, 0, 0, \dots] \tag{19}$$

Note that tableau size is determinable from content by summing: $|t| = \sum_x \mu(t)_x$. Also, content is additive over tableau multiplication: $\mu(tu) = \mu(t) + \mu(u)$, because tableau multiplication determines tu by re-arranging the entries of t and u into a larger tableau.

3.4.4 Shuffling

Given d and b, we can deduce that $\mu = \mu(a) = \mu(d) - \mu(b)$. We we can then pick the first word of search list w_1 as the word:

$$1^{\mu_1} 2^{\mu_2} \dots |d|^{\mu}_{|d|} \tag{20}$$

The remaining words in the word search list $[w_1, \dots, w_L]$ can be obtained by permuting the entries of r_1. Methods to generate all distinct permutations of a given word are well-known, and fast. The number of permutations of w_1, and thus the length L of the search list is then

$$L = \frac{(\mu_1 + \mu_2 + \mu_3 + \dots)!}{\mu_1! \mu_2! \mu_3! \dots!}. \tag{21}$$

This shuffling saves us from wasting time on trials a_i that have content inconsistent with d and b.

3.4.5 Shortening the list

The shuffling strategy will produce multiple words per tableau. This means that the list $[a_1, \dots, a_L]$ has repeated entries, even if the list of words $[w_1, \dots, w_L]$ has no repeats.

Given enough memory, some of the repeated computations $d_i = a_i b$ and $d_j = a_j b$ for $a_i = a_j$ can be avoided. Using Robinson–Schensted to get a_i from w_i is actually the first part of computing $d_i = a_i b$. A hash of a_i can be

stored in a list. When a_j is computed, it can be hashed, and compared to the list of hashes. It $a_j = a_i$ is detected via the hash list, then the rest of the computation $d_j = a_j b$ can be skipped.

There also exist various combinatorial methods to generate tableaus directly (such as [GNW79] and [NPS97], see [Sag01] for others). These generation methods might take longer to generate search list $a_1, \ldots, a_L$, and might arguably fall outside the category of trial multiplication. More importantly, it is unclear by how much they would speed up division.

3.4.6 Non-cancellation increases success rate

The length L of the search list $[a_1, \ldots, a_L]$ is an upper bound on the number of trial multiplications need.

Knuth multiplication is non-cancellative, so there would typically be many different a_i such that $a_i b = d$. If C is the average number of such a_i, then the L/C is a better estimate for the number of trial multiplications that will be needed.

As a speculative heuristic, an a priori guess is that $C \approx \sqrt{L}$ might measure the amount of non-cancellation, so that $\sqrt{L}$ trial multiplications would be needed. This has not been estimate empirically.

3.4.7 Estimates for the challenge

The number of shuffled words in the word list for the challenge division problem is approximately 2^{2851}. Under the speculative heuristic of non-cancellation, this mean at least 2^{1400} trial multiplications would be needed.

3.5 Division by max-algebra matrices

Johnson and Kambites [JK19] found a way to represent a semistandard tableau as a matrix, such that Knuth multiplication converts to matrix multiplication over a max-algebra. A max-algebra (also known as a tropical algebra), consists of numbers, but with different operations. The usual addition operation is replaced by maximization. (The usual multiplication operation is either: kept the same, in which case, only non-negative numbers are used; or in some versions, replaced by addition, in which case an extra number, $-\infty$, is included in the algebra.)

This suggests dividing tableaus can be achieved by dividing max-algebra matrices.

The Johnson–Kambites representation takes a tableau with entries in the set $\{1, \ldots, n\}$, and represents it as a square matrix with 2^n rows and 2^n columns. For large n, these matrices are quite large. Max-algebra division of 2^n by 2^n matrices can be done in about 8^n arithmetic operations on numbers, but the matrices in the Johnson–Kambites representation are sparse and trianglular, so might division might be be close to taking 2^n arithmetic operations.

These considerations suggest a tentative estimate that, for large size tableaus, division by erosion will be faster than using the Johnson–Kambites representation and max-algebra matrix division.

4 Key agreement

Alice and Charlie agree on a secret key as follows. All keys are semistandard tableaus. Alice generates her private key a; Charlie generates his private key c. Alice and Charlie both have the same base key b (but b is not secret). Alice computes her public key $d = ab$ and delivers d to Charlie. Charlie computes and delivers his public key $e = bc$ to Alice. Alice computes a secret key $f = ae$. Charlie computes a secret key $g = dc$. Alice and Charlie's secret keys f and g agree, because $f = abc = g$ and Knuth multiplication is associative, $a(bc) = (ab)c$.

In practice, Alice and Charlie will apply a key derivation function H to the agreed secret key f to derive a encryption or authentication key $k = H(f)$. Some reasons for this are that: f might be the wrong length to use as encryption or authentication key, f might be too easily distinguishable from a uniformly random byte string. Key derivation functions aim to address these issues, because they have variable length outputs, and are pseudorandom functions (secret inputs give outputs that look random).

4.1 Main security aims

The main security aim of plactic key agreement is to resist an attacker who tries to compute the agreed secret key $f = g$ by knowing the base key b, and the delivered keys d and e.

Note that several other types of attackers can be defined. For example, some attackers might not observe d, but can observe the use of the key f. Some attackers might not know b, perhaps because it is derived from a password, and will try to guess b. Some attackers might somehow get to see a set of possible agreed secret keys $\{f_1, f_2\}$ that contains f, and need to find i such that $f_i = f$. Resisting these other attackers is only secondary aim of plactic key agreement, and would not be too useful if the main security aim could not be met.

4.2 Attacking key agreement using division

The attacker can compute the agreed secret key f by the computation

$$f = (d/b)e. \tag{22}$$

The proof that this works is an elementary calculation:

$$\begin{aligned} f &= abc \\ &= (ab)c \\ &= ((ab)/b)b)c \\ &= ((d/b)b)c \\ &= (d/b)(bc) \\ &= (d/b)e. \end{aligned}$$

Note that when the attacker computes d/b, it is not necessarily the case that $a = d/b$, because the plactic monoid is non-cancellative. The value d/b can be considered, however, to be **effectively equivalent** to Alice's private key, because both private keys a and d/b generate the same public key d. In other words, $(d/b)b = ab$. In this sense, using division to attack plactic key agreement is similar to solving the discrete logarithm problem to attack elliptic curve Diffie–Hellman key agreement, because both attacks find the effective private key of one of the users.

4.3 Shell script for key agreement

A demo shell script[4] using the plactic monoid for key agreement is in Table 7.

[4]The script uses special features of the bash shell to arrange for the command lines to look somewhat like the standard notations in algebra, such as $d = ab$.

```
# demo.sh
=? () { head -c 384 /dev/urandom | base64 ; }
=  () { cat $@ | ./knuth_mult ; }
> a =?         # Alice:   secretly generate
> b =?         # Both:    jointly  generate
> c =?         # Charlie: secretly generate
> d = a b      # Alice:   openly deliver to Charlie
> e = b c      # Charlie: openly deliver to Alice
> f = a e      # Alice:   secretly compute
> g = d c      # Charlie: secretly compute
sha256sum f g  # Both:    hashed key ready to use in encryption, etc.
```

Table 7: Demo shell script for plactic key agreement

An example run of the demo script in Table 7 is included in Table 8, merely showing that Alice and Charlie compute the same hash. The values

```
$ ./demo.sh
0defbeb9f0e3e62fbe99b32db3325d837857078e6dfcf5537b6fab490462bf58  f
0defbeb9f0e3e62fbe99b32db3325d837857078e6dfcf5537b6fab490462bf58  g
```

Table 8: A run of plactic key agreement

of the files d and b from this run are shown in Tables 3 and 4. The file b is not put into semistandard tableau form.

Heuristically extrapolating empirical run-times of division-by-erosion, suggests that computing d/b using division-by-erosion should be expected to take at least 2^{128} steps (on average for d and b computed randomly of similar size).

5 Discussion

5.1 Provenance

The combinatorics underlying plactic key agreement has prominent provenance, originating from very reputable authors, such as Knuth and Schutzenberger. If a fast division algorithm was obvious to these authors, maybe they

had no reason to mention it.

5.2 Diversification

Plactic key agreement seems independent of the more established cryptography schemes such as: elliptic curve Diffie–Hellman (ECDH), supersingular isogeny key exchange (SIKE), as well as McEliece public-key encryption, and NTRU public key-encryption. It seem unlikely that an attack on one of these established schemes would extend to plactic key agreement.

5.3 Quantum computer attack vulnerability

The combinatorial nature of plactic key agreement involves many branching steps and non-reversible operations. Quantum computers tend to have advantage over classical computers when many non-branching and reversible operations can be computed in quantum superposition. This intuition suggests that plactic key agreement might have some innate immunity to quantum computer attacks. (On the other hand, perhaps the Johnson–Kambites matrix representation runs faster on a quantum computer.)

Acknowledgments

A. Menezes, A. Živković, and three anonymous reviewers provided helpful suggestions to improve the clarity of this report.

T. Suzuki supported and encouraged the proposal of plactic key agreement.

A Previous work

This section outlines the history of the technical elements making up plactic key agreement.

A.1 Semistandard tableaus and the plactic monoid

Kostka [Kos82] introduced semistandard tableaus to provide combinatorial explanations of the integer coefficients of symmetric polynomials. The coefficient of the monomial symmetric polynomials in the Schur symmetric

polynomial is the number of semistandard tableaus of a given shape and content (and now the numbers are called Kostka numbers). Jacobi [Jac41] defined Schur symmetric polynomials as ratios of alternating polynomials. Schur later used these polynomials in matrix representations of permutations [Sch01] (and now these polynomials are name after Schur). Young [You00] used standard tableaus (semistandard tableaus with all entries distinct) to study matrix representation of permutations (and now these tableaus are often called Young tableaus).

Robinson [Rob38] used semistandard tableaus in 1938 to find the longest non-decreasing subsequence of a given sequence. Schensted [Sch61] extended this method in 1961 to find longest subsequences that were unions of a given number non-decreasing subsequences. Their independently discovered algorithm is now called the Robinson–Schensted algorithm. It is the part of the Knuth multiplication that one gets from multiplying an empty tableau a by some other tableau b, except that one starts from the row reading of b. The shape of the tableau b provides information about various subsequences of b.

Knuth [Knu70] defined a monoid, which he called "tableau algebra". Its elements can be represented by semistandard tableaus. To prove associativity of multiplication of tableaus, Knuth showed that the Robinson–Schensted algorithm that maps strings to semistandard tableaus actually defines a congruence on the monoid of strings under concatenation (the free monoid). The resulting congruence monoid is the plactic monoid.

Lascoux and Schutzenberger [LS81] studied Knuth's "tableau algebra" in 1981 and re-named it *monoide plaxique*[5], which has been translated to **plactic monoid**, which is now the generally accepted term.

For a sample of some recent research about the plactic monoid, see [CM18] and [JK19].

A.2 Key agreement

Diffie and Hellman [DH76] introduced in 1976 a **key agreement** scheme[6]. Merkle [Mer78] independently introduced another (less efficient) type of key agreement. Diffie–Hellman key agreement uses exponentiation modulo a large prime number. Changing their notation slightly, Alice sends $d = b^a$

[5]The new name may be inspired by plate tectonics.

[6]Diffie and Hellman refer to their key agreement scheme as a "public-key distribution system". This report uses the term **key agreement** to avoid clashes with the more general meanings of key distribution and key exchange.

to Charlie, Charlie sends $e = b^c$ to Alice, Alice computes agreed key $f = e^a$, and Charlie computes agreed key $g = d^c$.

Berenstein and Chernyak [BC04] described in 2004 a key agreement scheme that uses semigroup multiplications[7] instead of modular exponentiation.

References

[BC04] Arkady Berenstein and Leon Chernyak. Geometric key establishment. In *Canadian Mathematical Society Conference*, pages 1–19, 2004. 1, A.2

[CM18] Alan J. Cain and António Malheiro. Identities in plactic, hypoplactic, Sylvester, Baxter, and related monoids. *The Electronic Journal of Combinatorics*, 25(3), August 2018. A.1

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, November 1976. 1, A.2

[GNW79] C. Greene, A. Nijenhuis, and H. S. Wilf. A probabilistic proof of a formula for the number of Young tableaux of a given shape. *Adv. in Math.*, 31:104–109, 1979. 3.4.5

[Jac41] Carl Gustav Jacobi. De functionibus alternantibus. *Crelle's Journal*, 22:360–371, 1841. A.1

[JK19] Marianne Johnson and Mark Kambites. Tropical representations and identities of plactic monoids, 2019. 1, 3.5, A.1

[Knu70] Donald E. Knuth. Permutations, matrices, and generalized Young tableaux. *Pacific Journal of Mathematics*, 34(3):709 – 727, 1970. 1, 2, 2.1, 2.5, 3.3.3, A.1

[Kos82] Carl Kostka. Über den zusammenhang zwischen einigen formen von symmetrischen funktionen. *Crelle's Journal*, 93:89–123, 1882. A.1

[7] A semigroup is a set with an associative binary operation. By default, we assume a multiplicative notation for the binary operation. A monoid is a semigroup that has an identity element. In particular, the plactic monoid is a semigroup.

[LS81] Alain Lascoux and Marcel P. Schützenberger. Le monoïde plaxique. In *Proc. Colloqu. Naples, Noncommutative structures in algebra and geometric combinatorics (Naples, 1978)*, volume 109 of *Quad. Ricerca Sci.*, pages 129 – 156. CNR, Rome 1981. 2.5, A.1

[Mer78] Ralph C. Merkle. Secure communications over insecure channels. *Communications of the ACM*, 21(4):294–299, April 1978. A.2

[NPS97] J. C. Novelli, I. M. Pak, and A. V. Stoyanovskii. A direct bijective proof of the hook-lengthe formula. *Discrete Math. Theoret. Computer Science*, 1:53–67, 1997. 3.4.5

[Rob38] Gilbert de B. Robinson. On the representations of the symmetric group. *Amer. J. Math.*, 60, 1938. 2.6, A.1

[Sag01] Bruce E. Sagan. *The Symmetric Group: Representations, Combinatorial Algorithms, and Symmetric Functions*. Number 203 in Graduate Texts in Mathematics. Springer, 2nd edition, 2001. 3.4.5

[Sch01] Issai Schur. *Über eine Klasse von Matrizen, die sich einer gegebenen Matrix zuordnen lassen. Doctoral dissertation*. PhD thesis, Universität Berlin, 1901. A.1

[Sch61] Craige Schensted. Longest increasing and decreasing subsequences. *Canad. J. Math.*, 13:179–191, 1961. 2.6, A.1

[You00] Alfred Young. On quantitative substitutional analysis. *Proceedings of the London Mathematical Society, Ser. 1*, 33(1):97–145, 1900. A.1