Optimizing linear maps modulo 2

Daniel J. Bernstein [⋆]

Department of Computer Science (MC 152)
The University of Illinois at Chicago
Chicago, IL 60607–7053
`djb@cr.yp.to`

Abstract. This paper introduces and analyzes an algorithm to compile a series of exclusive-or operations. The compiled series is quite efficient, almost always beating the so-called "Four Russians" approach, and uses no temporary storage beyond its outputs. The algorithm is reasonably fast and surprisingly simple.

1 Introduction

Consider the 4-bit-to-4-bit $\mathbf{F}_2$-linear function $L : \mathbf{F}_2^4 \to \mathbf{F}_2^4$ defined by $L(x_0, x_1, x_2, x_3) = (x_0 \oplus x_1 \oplus x_2 \oplus x_3, x_0 \oplus x_2 \oplus x_3, x_0 \oplus x_1 \oplus x_2, x_0 \oplus x_3)$; i.e.,

$$L\begin{pmatrix}x_0\\x_1\\x_2\\x_3\end{pmatrix} = \begin{pmatrix}1&1&1&1\\1&0&1&1\\1&1&1&0\\1&0&0&1\end{pmatrix}\begin{pmatrix}x_0\\x_1\\x_2\\x_3\end{pmatrix}.$$

Evaluating L directly from its definition takes 8 bit xors: xor x_0 and x_1, then xor x_2, then xor x_3, obtaining $(Lx)_0$; xor x_0 and x_2, then xor x_3, obtaining $(Lx)_1$; xor x_0 and x_1, then xor x_2, obtaining $(Lx)_2$; finally xor x_0 and x_3, obtaining $(Lx)_3$. However, this computation has several obvious redundancies, and a closer look shows that L can be computed in just 4 xors:

$$\begin{tikzcd}
(Lx)_0 & & \oplus \arrow[ll] \arrow[ddl] & & & x_0 \arrow[dddl, bend right] \\
(Lx)_1 & & & \oplus \arrow[lll] \arrow[ul] & & x_1 \arrow[ulll, bend right] \\
(Lx)_2 & \oplus \arrow[l] & & & & x_2 \arrow[ull, bend right] \\
(Lx)_3 & & & & \oplus \arrow[llll] \arrow[uul] & x_3 \arrow[l] \arrow[ullll, bend right]
\end{tikzcd}$$

[⋆] Permanent ID of this document: `e5c3095f5c423e2fe19fa072e23bd5d7`. Date of this document: 2009.08.30. This work was supported by the National Science Foundation under grant ITR–0716498.

Similarly, if x_0, x_1, x_2, x_3 are (e.g.) 128-bit vector registers, then the 8 vector xors in $(x_0 \oplus x_1 \oplus x_2 \oplus x_3, x_0 \oplus x_2 \oplus x_3, x_0 \oplus x_1 \oplus x_2, x_0 \oplus x_3)$ can be replaced by 4 vector xors.

This paper presents an algorithm that, given the matrix for a p-bit-to-q-bit $\mathbf{F}_2$-linear function $L : \mathbf{F}_2^p \to \mathbf{F}_2^q$, prints code to compute L. Heuristics for large p, q suggest, and experiments confirm, that for most functions L the code uses approximately $pq/(\lg q - \lg\lg q)$ xors. The code writes only to the q output bits and does not need any extra storage; it does not require tradeoffs between space and time. The code reads each of the p input bits exactly once, in order, except that it skips unused bits. The compilation algorithm per se is reasonably fast and surprisingly simple.

Applications. Bitsliced binary-finite-field arithmetic has recently set software speed records for public-key cryptography; see [5]. The software described in [5] spends most of its time in short hand-optimized linear computations such as

```
vec h0 = h[i];
vec h12 = h[i + n] ^ h[i + 2 * n];
vec h34 = h[i + 3 * n] ^ h[i + 4 * n];
vec h56 = h[i + 5 * n] ^ h[i + 6 * n];
vec h7 = h[i + 7 * n];
vec x0 = u[i];
vec x12 = u[i + n] ^ u[i + 2 * n];
vec x3 = u[i + 3 * n];
vec h1 = h12 ^ h0 ^ u2[i];
vec b = h34 ^ h12 ^ u2[i + n];
vec c = h34 ^ h56 ^ u3[i];
vec h6 = h7 ^ h56 ^ u3[i + n];
h[i + n] = h1;
h[i + 2 * n] = b ^ h0 ^ x0;
h[i + 3 * n] = c ^ h1 ^ x12 ^ x0 ^ u4[i];
h[i + 4 * n] = h6 ^ b ^ x12 ^ x3 ^ u4[i + n];
h[i + 5 * n] = h7 ^ c ^ x3;
h[i + 6 * n] = h6;
```

where each ^ is a 128-bit vector xor. Unfortunately, hand optimization is time-consuming even for small examples.

This paper's algorithm can be used as a baseline compilation technique for all $\mathbf{F}_2$-linear functions L. In many cases the algorithm is competitive with the best hand-optimized code, saving time for the programmer. The algorithm can also be applied to extremely large examples: for example, I have used it to generate fast unrolled bitsliced normal-basis conversion functions for the cryptanalytic computation described in [3], converting a 131×131 basis-conversion matrix into a sequence of 3380 xors and converting a 163×163 matrix into a sequence of 5078 xors.

This paper focuses on the number of xors as a simplified model of software time. The model is reasonably accurate for computations that fit into registers: the code does not require three-operand instructions, and it usually has enough parallelism to occupy all available arithmetic units. The same algorithm also appears to perform reasonably well for larger computations, but a detailed analysis of load/store costs is beyond the scope of this paper.

One should not think of this algorithm as magically discovering every fast linear computation in the literature. For example, if $\varphi \in \mathbf{F}_2[x]$ is a polynomial of degree n, then the $\mathbf{F}_2$-linear function $f \mapsto f^2$ on $\mathbf{F}_2[x]/\varphi$, with basis $1, x, \ldots, x^{n-1}$, can be computed with $O(n \lg n)$ xors by fast-multiplication techniques. For most choices of φ, feeding the same linear function to this paper's algorithm would use many more xors, at least for large n.

Previous methods. "Input partitioning" is the following classic method to compute q sums of subsequences of $x_0, x_1, \ldots, x_{p-1}$:

- Partition $\{0, 1, \ldots, p-1\}$ into p/c parts of size c. This description assumes for simplicity that p is a multiple of c.
- For each part, and for each nonempty subset S of the part, compute the subset sum $\sum_{i \in S} x_i$. This takes $(p/c)(2^c - c - 1)$ additions; note that each new subset sum requires only one addition.
- Compute each of the output sums as a sum of (at worst) p/c subset sums. This takes $q(p/c-1)$ additions. Note that the speedup here goes beyond common-subexpression elimination; it takes advantage of the associativity of addition.

A standard analysis—disregarding variants such as eliminating unused subsets—chooses $c \in \lg q - \lg \lg q - \lg \log 2 + o(1)$ and produces a bound of $pq(1+(1/\log 2+\lg\log 2+o(1))/\lg q)/(\lg q-\lg\lg q)$ additions. The total number of additions is therefore asymptotically $(1 + o(1))q^2/\lg q$ in the typical case $p = q$.

Input partitioning for vectors of Boolean truth values was introduced by Lupanov in [10]. Obviously input partitioning also works for vectors of integers, vectors of integers modulo 2, etc. Fifteen years later the same construction appeared in [2, "Lemma (M. Kronrod)"] as a tool for Boolean matrix multiplication. Input partitioning is often called the "Four-Russians algorithm" by people who

- see that [2] was written by Arlazarov, Dinic, Kronrod, and Faradžev, all of which sound like Russian names to the ignorant observer;
- have not actually read [2], and are thus unaware that the method is credited to Kronrod alone;
- are unaware of previous work such as [10]; and
- are unaware that normal scientific standards require giving credit by name.

See, e.g., [15], [11], [4], and [6].

An extra technique, which I call "output clumping," asymptotically reduces $(1+o(1))q^2/\lg q$ to $(1+o(1))q^2/\lg(q^2)$, saving a factor of $2+o(1)$. This technique was introduced by Nechiporuk and pushed much further by Pippenger; see [12], [13], and [14]. Pippenger's addition chains are within a factor of $1+o(1)$ of optimal for a wide variety of problems; see generally [13, Section 2] and [14, Section 2].

Trifonov in [16] states that input partitioning produces addition chains of length $2q^2/\lg q$, and that a much slower—but still usable—algorithm finds "considerably smaller" xor chains. Input-partitioning chains actually have length $(1+o(1))q^2/\lg q$, so the comparison in [16] is clearly erroneous; it is nevertheless possible that the algorithm of [16] has some merit.

Algorithm comparison. This paper's algorithm produces xor chains that, for $p = q$, have length $(1+o(1))q^2/\lg q$. Input partitioning also produces chains of length $(1+o(1))q^2/\lg q$, but a closer look at the $o(1)$ suggests that this paper's chains are shorter by a factor of nearly $1+1/\lg q$. Input partitioning also appears to be somewhat less memory-friendly than this paper's algorithm.

Pippenger's addition chains are shorter than this paper's chains for sufficiently large p, q. However, preliminary experiments indicate that this paper's xor chains are shorter than Pippenger's addition chains for $p, q \leq 256$. Perhaps there is some way to combine the ideas of these algorithms.

2 The algorithm

An efficient multi-scalar-multiplication method appears in [7, Section 4] with credit to Bos and Coster. To compute $n_0x_0 + n_1x_1 + n_2x_2 + \cdots$, where $n_0 \geq n_1 \geq n_2 \geq \cdots \geq 0$, Bos and Coster recursively compute $(n_0 - n_1)x_0 + n_1(x_0 + x_1) + n_2x_2 + \cdots$. They use a more complicated step in the case that n_0 is much larger than n_1, since subtracting n_1 from

n_0 is then ineffective at reducing n_0, although this case rarely occurs for random scalars.

A transposed version of the Bos–Coster method computes multiples $n_0x, n_1x, n_2x, \ldots$, where $n_0 \geq n_1 \geq n_2 \geq \cdots$, by recursively computing $(n_0 - n_1)x, n_1x, \ldots, n_{q-1}x$ and then adding output 1 into output 0.

This paper's algorithm has the same outline but uses xor instead of subtraction: it computes several dot products $L_0x, L_1x, L_2x, \ldots$, where $L_0 \geq L_1 \geq L_2 \geq \cdots$, by recursively computing $(L_0 \oplus L_1)x, L_1x, \ldots, L_{q-1}x$ and then xoring output 1 into output 0. The case that L_0 is much larger than L_1 (specifically, that it has its most significant bit at a different position) is much more common here, and requires different treatment: the algorithm reduces L_0 by simply clearing its most significant bit.

Details. The input to the algorithm is a $q \times p$ matrix of bits, viewed as a sequence of q rows $L_0, L_1, \ldots, L_{q-1} \in \mathbf{F}_2^p$, where p and q are non-negative integers. The p bits $L_j[0], L_j[1], \ldots, L_j[p-1]$ of L_j specify the linear function $x_0, x_1, \ldots, x_{p-1} \mapsto L_j[0]x_0 \oplus L_j[1]x_1 \oplus \cdots \oplus L_j[p-1]x_{p-1}$; the algorithm produces code that computes these q linear functions. The algorithm works as follows:

- If $q = 0$: Stop. (There is nothing to compute.)
- If $p = 0$: Generate code that sets each output bit to 0. Stop.
- Find $j \in \{0, 1, \ldots, q-1\}$ that maximizes L_j in reverse lexicographic order (i.e., maximizes $L_j[p-1]$; secondarily maximizes $L_j[p-2]$; etc.).
- If $L_j[p-1] = 0$: Define L'_k as $(L_k[0], \ldots, L_k[p-2])$ for each $k \in \{0, 1, \ldots, q-1\}$. Recursively apply the algorithm to $L'_0, L'_1, \ldots, L'_{q-1}$. Stop. (All $L_k[p-1]$ are 0; i.e., x_{p-1} is unused.)
- If $q \geq 2$: Find $i \in \{0, 1, \ldots, q-1\} - \{j\}$ that maximizes L_i in reverse lexicographic order. If $L_i[p-1] = 1$: Define $L'_k = L_k$ for each $k \in \{0, 1, \ldots, q-1\}$, except that $L'_j = L_j \oplus L_i$. Recursively apply the algorithm to $L'_0, L'_1, \ldots, L'_{q-1}$. Generate code that xors output bit i into output bit j. Stop.
- Define $L'_k = L_k$ for each $k \in \{0, 1, \ldots, q-1\}$, except that $L'_j[p-1] = 0$. Recursively apply the algorithm to $L'_0, L'_1, \ldots, L'_{q-1}$. Generate code that xors *input* bit $p-1$ into output bit j. Stop.

The recursive steps in the algorithm can and should store L' on top of L, eliminating the space for L' and almost all of the time to compute L'. If the output code is generated in reverse order then the recursive steps can be replaced by tail-recursive steps, eliminating all other storage. If the rows, or pointers to the rows, are stored in a heap then identifying the two largest rows takes only a logarithmic number of row comparisons.

The code generated by this algorithm starts with code to set each output bit to 0. Except in the extreme case $L_j = 0$, the code to set output bit j to 0 can and should be merged with a subsequent xor involving output bit j, turning the xor into a copy. One can, as an option, merge the copies with further xors, although in some cases this is incompatible with reading each input exactly once.

A transposed version of the same algorithm writes each of the output bits exactly once, in order, and otherwise works entirely within the p input bits.

Example. Consider the four rows appearing at the start of this paper: $L_0 = (1\,1\,1\,1)$; $L_1 = (1\,0\,1\,1)$; $L_2 = (1\,1\,1\,0)$; $L_3 = (1\,0\,0\,1)$.

The largest row in reverse lexicographic order is $L_0 = (1\,1\,1\,1)$, and the second largest is $L_1 = (1\,0\,1\,1)$. The last instruction in the compiled code is to xor output bit 1 into output bit 0. The goal of the previous instructions is to compute $L'_0 = (0\,1\,0\,0)$; $L'_1 = (1\,0\,1\,1)$; $L'_2 = (1\,1\,1\,0)$; $L'_3 = (1\,0\,0\,1)$. Here $L'_0 = L_0 \oplus L_1$.

The largest remaining row is $L'_1 = (1\,0\,1\,1)$, followed by $L'_3 = (1\,0\,0\,1)$. The second-to-last instruction in the compiled code is to xor output bit 3 into output bit 1. The goal of the previous instructions is to compute $L''_0 = (0\,1\,0\,0)$; $L''_1 = (0\,0\,1\,0)$; $L''_2 = (1\,1\,1\,0)$; $L''_3 = (1\,0\,0\,1)$. Here $L''_1 = L'_1 \oplus L'_3$.

The largest remaining row is $L''_3 = (1\,0\,0\,1)$, followed by $L''_2 = (1\,1\,1\,0)$. The third-to-last instruction in the compiled code is to xor *input* bit 3 into output bit 3. The goal of the previous instructions is to compute $L'''_0 = (0\,1\,0)$; $L'''_1 = (0\,0\,1)$; $L'''_2 = (1\,1\,1)$; $L'''_3 = (1\,0\,0)$.

The largest remaining row is $L'''_2 = (1\,1\,1)$, followed by $L'''_1 = (0\,0\,1)$. The fourth-to-last instruction in the compiled code is to xor output bit 1 into output bit 2. Et cetera. The algorithm finishes with the following sequence of instructions:

- Store 0 in output bit 0.
- Store 0 in output bit 1.
- Store 0 in output bit 2.
- Store 0 in output bit 3.
- Xor input bit 0 into output bit 3.
- Xor output bit 3 into output bit 2.
- Xor input bit 1 into output bit 0.
- Xor output bit 0 into output bit 2.
- Xor input bit 2 into output bit 1.
- Xor output bit 1 into output bit 2.

- Xor input bit 3 into output bit 3.
- Xor output bit 3 into output bit 1.
- Xor output bit 1 into output bit 0.

Eliminating 0 produces the following sequence of instructions:

- Copy input bit 0 into output bit 3.
- Copy output bit 3 into output bit 2.
- Copy input bit 1 into output bit 0.
- Xor output bit 0 into output bit 2.
- Copy input bit 2 into output bit 1.
- Xor output bit 1 into output bit 2.
- Xor input bit 3 into output bit 3.
- Xor output bit 3 into output bit 1.
- Xor output bit 1 into output bit 0.

One can, as an option, eliminate the copies:

- Xor input bit 1 and input bit 0, producing output bit 2.
- Xor input bit 2 into output bit 2.
- Xor input bit 3 to input bit 0, producing output bit 3.
- Xor output bit 3 to input bit 2, producing output bit 1.
- Xor output bit 1 to input bit 1, producing output bit 0.

3 Experimental results

Appendix A shows `xor.cpp`, a straightforward C++ implementation of this paper's algorithm. Running

```
g++ -o xor xor.cpp
echo 1111101111101001 | ./xor 4 4 > test.c
gcc -o test test.c
./test
```

applies the algorithm to the function L shown at the beginning of this paper; creates a test program `test.c` for the resulting code; and prints 4 zeros, indicating that all 4 outputs were computed correctly.

More generally, `xor` p q reads $q \times p$ input bits, applies this paper's algorithm, and prints a test program that prints q zeros. I checked this for 10 random $q \times p$ matrices (from `/dev/urandom`, the Linux cryptographic random number generator) for each pair $(p, q) \in \{1, 2, 3, \ldots, 10\}^2$. The program packs each matrix row into an `unsigned long long`, so it is limited to $p \in \{0, 1, 2, \ldots, 64\}$, but it allows any $q \in \{1, 2, 3, \ldots\}$ that fits into memory.

The output of `xor` begins with two comment lines stating (1) the cost of the computation and (2) the cost after zero elimination. I used a separate script to check, again for 10 random matrices of each size, that the number of xors in the code matched (1) the first comment and matched (2) the second comment plus the number of nonzero L rows.

The following table shows the average cost after zero elimination, divided by pq, for 10000 random $q \times p$ matrices:

	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$
$q = 1$	0.0000	0.1242	0.2638	0.3785	0.4393	0.4691	0.4845
$q = 2$	0.0000	0.1118	0.2352	0.3407	0.3987	0.4293	0.4434
$q = 4$	0.0000	0.0866	0.2041	0.3019	0.3573	0.3860	0.4002
$q = 8$	0.0000	0.0562	0.1647	0.2603	0.3151	0.3424	0.3564
$q = 16$	0.0000	0.0309	0.1209	0.2158	0.2694	0.2965	0.3101
$q = 32$	0.0000	0.0156	0.0769	0.1721	0.2247	0.2514	0.2647
$q = 64$	0.0000	0.0078	0.0424	0.1350	0.1859	0.2119	0.2250
$q = 128$	0.0000	0.0039	0.0215	0.1055	0.1537	0.1794	0.1922
$q = 256$	0.0000	0.0020	0.0107	0.0789	0.1282	0.1530	0.1657
$q = 512$	0.0000	0.0010	0.0054	0.0525	0.1066	0.1318	0.1443
$q = 1024$	0.0000	0.0005	0.0027	0.0296	0.0868	0.1140	0.1269
$q = 2048$	0.0000	0.0002	0.0013	0.0151	0.0730	0.1003	0.1125
$q = 4096$	0.0000	0.0001	0.0007	0.0075	0.0652	0.0886	0.1006

The following table shows the standard deviation of the same quantity for the same matrices:

	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$
$q = 1$	0.0000	0.2161	0.2227	0.1736	0.1240	0.0885	0.0621
$q = 2$	0.0000	0.1243	0.1322	0.1026	0.0731	0.0515	0.0361
$q = 4$	0.0000	0.0577	0.0763	0.0610	0.0423	0.0301	0.0214
$q = 8$	0.0000	0.0188	0.0421	0.0347	0.0243	0.0172	0.0120
$q = 16$	0.0000	0.0031	0.0198	0.0186	0.0128	0.0092	0.0064
$q = 32$	0.0000	0.0002	0.0072	0.0100	0.0070	0.0048	0.0034
$q = 64$	0.0000	0.0000	0.0014	0.0060	0.0040	0.0027	0.0019
$q = 128$	0.0000	0.0000	0.0001	0.0040	0.0023	0.0016	0.0011
$q = 256$	0.0000	0.0000	0.0000	0.0024	0.0015	0.0009	0.0006
$q = 512$	0.0000	0.0000	0.0000	0.0011	0.0009	0.0006	0.0004
$q = 1024$	0.0000	0.0000	0.0000	0.0002	0.0004	0.0003	0.0002
$q = 2048$	0.0000	0.0000	0.0000	0.0000	0.0003	0.0002	0.0001
$q = 4096$	0.0000	0.0000	0.0000	0.0000	0.0002	0.0002	0.0001

For example, this algorithm evaluates 64-bit-to-128-bit linear maps using approximately $0.1922 \cdot 128 \cdot 64 \approx 1575$ xors on average, with standard deviation approximately $0.0011 \cdot 128 \cdot 64 \approx 9$.

For comparison, consider partitioning 64 inputs into 16 4-bit parts, and computing 128 outputs from xors of subsets of the parts. One expects each of the 128 outputs to involve $16(1-1/2^4)$ nonzero subsets on average, and therefore to consume at least $16(1-1/2^4)-1$ xors on average. Each part has 2^4-4-1 subsets of size 2 or larger; each subset is needed with probability $1-(1-1/2^4)^{128}$, and therefore consumes on average at least $1-(1-1/2^4)^{128}$ xors. The total cost is at least $128(16(1-1/2^4)-1)+16(2^4-4-1)(1-(1-1/2^4)^{128}) \approx 1967.95$ xors on average.

A better strategy is to partition 64 inputs into 10 6-bit parts and 1 4-bit part. The total cost is then at least $128(10(1-1/2^6)+1(1-1/2^4)-1)+10(2^6-6-1)(1-(1-1/2^6)^{128})+1(2^4-4-1)(1-(1-1/2^4)^{128}) \approx 1757.06$ xors on average.

One can consider other strategies, including "fractional" possibilities such as partitioning 64 inputs into 4 6-bit parts and 8 5-bit parts, but none of these strategies seem to come close to the 1575 xors used by the algorithm introduced in this paper.

Heuristic analysis. To understand the performance of this algorithm for $q = 128$, assume that exactly 64 of the 128 input rows involve the most significant input bit x_{p-1}. Performing 64 xors of adjacent rows typically produces 6 or more clear bits in each new row:

- At most 1 of the new rows can start with x_{p-2}: scanning through the sorted rows produces only one transition from $\ldots,0,1$ to $\ldots,1,1$.
- At most 2 of the new rows can start with x_{p-3}: one for the transition from $\ldots,0,1,1$ to $\ldots,1,1,1$, and one for the transition from $\ldots,0,0,1$ to $\ldots,1,0,1$.
- At most 4 of the new rows can start with x_{p-4}.
- At most 8 of the new rows can start with x_{p-5}.
- At most 16 of the new rows can start with x_{p-6}.
- The remaining 33 new rows must start with x_{p-7} or beyond.

At this point there are about 33 rows starting with x_{p-2}, and the next 33 xors typically produce 5 or more clear bits. After a few more iterations the algorithm settles down on a steady state consuming fewer than 30 xors for each bit.

For general q, the steady state appears to have approximately q/c rows starting with the most significant bit, where $c \in \mathbf{R}$ satisfies $2^c = q/c$. The algorithm thus uses approximately $pq/(\lg q - \lg\lg q)$ xors.

References

[1] — (no editor), *17th annual symposium on foundations of computer science*, IEEE Computer Society, Long Beach, California, 1976. MR 56:1766. See [3].

[2] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, I. A. Faradžev, *On economical construction of the transitive closure of an oriented graph*, Soviet Mathematics Doklady **11** (1970), 1209–1210. ISSN 0197–6788. MR 42:4441. URL: http://cr.yp.to/bib/entries.html#1970/arlazarov. Citations in this document: §1, §1, §1.

[3] Daniel V. Bailey, Brian Baldwin, Lejla Batina, Daniel J. Bernstein, Peter Birkner, Joppe W. Bos, Gauthier van Damme, Giacomo de Meulenaer, Junfeng Fan, Tim Gneysu, Frank Gurkaynak, Thorsten Kleinjung, Tanja Lange, Nele Mentens, Christof Paar, Francesco Regazzoni, Peter Schwabe, Leif Uhsadel, *The Certicom challenges ECC2-X*, in Workshop Record of SHARCS'09 (2009). Citations in this document: §1.

[4] Gregory Bard, *Accelerating cryptanalysis with the Method of Four Russians* (2006). URL: http://eprint.iacr.org/2006/251. Citations in this document: §1.

[5] Daniel J. Bernstein, *Batch binary Edwards*, in [9] (2009), 317–336. Citations in this document: §1, §1.

[6] Tomas J. Boothby, Robert W. Bradshaw, *Bitslicing and the method of four Russians over larger finite fields*. URL: http://arxiv.org/abs/0901.1413. Citations in this document: §1.

[7] Peter de Rooij, *Efficient exponentiation using precomputation and vector addition chains*, in [8] (1995), 389–399. MR 1479665. Citations in this document: §2.

[8] Alfredo De Santis (editor), *Advances in cryptology: EUROCRYPT '94*, Lecture Notes in Computer Science, 950, Springer, Berlin, 1995. ISBN 3–540–60176–7. MR 98h:94001. See [3].

[9] Shai Halevi (editor), *Advances in Cryptology—CRYPTO 2009, 29th annual international cryptology conference, Santa Barbara, CA, USA, August 16–20, 2009, proceedings*, Lecture Notes in Computer Science, 5677, Springer, 2009. See [3].

[10] O. B. Lupanov, *On rectifier and contact-rectifier circuits*, Doklady Akademii Nauk SSSR **111** (1956), 1171–1174. ISSN 0002–3264. URL: http://cr.yp.to/bib/entries.html#1956/lupanov. Citations in this document: §1, §1.

[11] Eugene W. Myers, *A four Russians algorithm for regular expression pattern matching*, Journal of the ACM **39** (1992), 430–448. Citations in this document: §1.

[12] Nicholas Pippenger, *On the evaluation of powers and related problems (preliminary version)*, in [1] (1976), 258–263; newer version split into [13] and [14]. MR 58:3682. URL: http://cr.yp.to/bib/entries.html#1976/pippenger. Citations in this document: §1.

[13] Nicholas Pippenger, *The minimum number of edges in graphs with prescribed paths*, Mathematical Systems Theory **12** (1979), 325–346; see also older version [12]. ISSN 0025–5661. MR 81e:05079. URL: http://cr.yp.to/bib/entries.html#1979/pippenger. Citations in this document: §1, §1.

[14] Nicholas Pippenger, *On the evaluation of powers and monomials*, SIAM Journal on Computing **9** (1980), 230–250; see also older version [12]. ISSN 0097–5397. MR 82c:10064. URL: http://cr.yp.to/bib/entries.html#1980/pippenger. Citations in this document: §1, §1.

[15] Nicola Santoro, *Extending the four Russians' bound to general matrix multiplication*, Information Processing Letters **10** (1980), 87–88. Citations in this document: §1.

[16] Peter Trifonov, *Matrix-vector multiplication via erasure decoding* (2007). URL: http://dcn.infos.ru/~petert/. Citations in this document: §1, §1, §1.

Appendix A: `xor.cpp`

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

typedef unsigned long long uint64;

uint64 *L;
uint64 *origL;

void L_read(int p,int q)
{
  for (int i = 0;i < q;++i)
    for (int j = 0;j < p;++j) {
      char c;
      cin >> c;
      L[i] = L[i] + (((uint64) (c & 1)) << j);
    }
}

int L_cmp(int a,int b) { return L[a] < L[b]; }

vector<int> target;
vector<int> source;
vector<const char *> sourcetype;

void L_compile(int p,int q)
{
  for (int i = 0;i < q;++i) origL[i] = L[i];
  int pos[q];
  for (int i = 0;i < q;++i) pos[i] = i;
  make_heap(pos,pos + q,L_cmp);
  while (p > 0) {
    pop_heap(pos,pos + q,L_cmp);
    if (L[pos[q - 1]] >> (p - 1))
      if (q >= 2 && (L[pos[0]] >> (p - 1))) {
        L[pos[q - 1]] ^= L[pos[0]];
        target.push_back(pos[q - 1]);
        sourcetype.push_back("out");
        source.push_back(pos[0]);
      } else {
        L[pos[q - 1]] ^= (((uint64) 1) << (p - 1));
        target.push_back(pos[q - 1]);
        sourcetype.push_back("in");
        source.push_back(p - 1);
      }
    else
      --p;
    push_heap(pos,pos + q,L_cmp);
  }
}

void print(int p,int q)
{
  int i;
  int cost = target.size();
  cout << "/* cost " << cost << " */\n";
  for (i = 0;i < q;++i) if (origL[i] != 0) --cost;
  cout << "/* cost " << cost << " after 0 elimination */\n";
  cout << "#include <stdio.h>\n";
  cout << "main() {\n";
  for (i = 0;i < p;++i)
    cout << "unsigned long long in" << i << " = "
         << (((uint64) 1) << i) << "ULL;\n";
  for (i = 0;i < q;++i)
    cout << "unsigned long long out" << i << " = 0;\n";
  i = target.size();
  while (i > 0) {
    --i;
    cout << "out" << target[i] << " ^= "
      << sourcetype[i] << source[i] << ";\n";
  }
  for (i = 0;i < q;++i)
    cout << "printf(\"%llu\\n\",out"
         << i << " ^ " << origL[i] << "ULL);\n";
  cout << "}\n";
}
```

```
int main(int argc,char **argv)
{
  int p = 8;
  int q = 8;
  if (argv[1]) {
    p = q = atoi(argv[1]);
    if (argv[2])
      q = atoi(argv[2]);
  }

  L = new uint64[q];
  origL = new uint64[q];

  L_read(p,q);
  L_compile(p,q);
  print(p,q);
  return 0;
}
```