Draft. I want to rethink all the notation. Taking logs would have some advantages.

PIPPENGER'S EXPONENTIATION ALGORITHM

DANIEL J. BERNSTEIN

Abstract. Pippenger's exponentiation algorithm computes a power, or a product of powers, or a sequence of powers, or a sequence of products of powers, with very few multiplications. Pippenger's algorithm was published twenty-five years ago, but it is still not widely understood or appreciated, although certain parts of it have recently been reinvented, republished, and popularized. This paper is an exposition of the state of the art in generic exponentiation algorithms: in particular, Pippenger's algorithm.

1. Introduction

One can compute the power x^{27182} with 18 multiplications; or the power x^{31415} with 19 multiplications; or the pair x^{31415}, x^{27182} with 21 multiplications; or the product $x_1^{31415}x_2^{27182}$ with 22 multiplications.

This paper explains the state of the art in algorithms that use multiplications (and nothing but multiplications!) to compute powers; more generally, to compute products of powers, or sequences of powers, or sequences of products of powers. Here is a summary of the major breakthroughs:

- Brauer's algorithm, published in 1939, computes a power. It uses at most about $(1 + 1/\lg\lg B)\lg B$ multiplications if the exponent is below B.
- Straus's algorithm, published in 1964, computes a product of p powers. It uses at most about $(1 + p/\lg\lg B)\lg B$ multiplications if each exponent is below B.
- Yao's algorithm, published in 1976, computes a sequence of p powers of a single base. It uses at most about $(1+p/\lg\lg B)\lg B$ multiplications if each exponent is below B.
- Pippenger's algorithm, published in 1976, replaces $\lg\lg B$ with $\lg(p\lg B)$, improving on both Straus's algorithm and Yao's algorithm when p is large. More generally, Pippenger's algorithm computes a sequence of q products of powers of p inputs. It uses at most about $(\min\{p, q\} + pq/\lg(pq\lg B))\lg B$ multiplications if each exponent is below B.

Section 2 of this paper explains a popular language for describing exponentiation algorithms. Section 3 explains Brauer's algorithm, several minor improvements pointed out by Knuth and Thurber, and Straus's algorithm. Section 4 explains Yao's algorithm. Section 5 explains the relationship between Brauer-Straus-type algorithms and Yao-type algorithms. Section 6 explains a special case of Pippenger's algorithm, where all the exponents are in $\{0, 1\}$. Section 7 explains the general case.

Date: 20020118.

1991 *Mathematics Subject Classification.* Primary 11Y16.

2. Addition chains

An **addition chain of length ℓ** is a sequence of $\ell + 1$ integers, where the first integer is 1 and each subsequent integer is a sum of two earlier integers. In other words, it is a sequence $c_0, c_1, \ldots, c_\ell$ such that $c_0 = 1$ and, for each $k \in \{1, 2, \ldots, \ell\}$, there exist $i, j \in \{0, 1, \ldots, k-1\}$ such that $c_k = c_i + c_j$.

For example, $1, 2, 3, 5, 7, 14, 28, 56, 63$ is an addition chain of length 8, because $2 = 1 + 1$, $3 = 2 + 1$, $5 = 3 + 2$, $7 = 5 + 2$, $14 = 7 + 7$, $28 = 14 + 14$, $56 = 28 + 28$, and $63 = 56 + 7$.

Using chains to compute powers. Short addition chains are a model of fast algorithms for modular exponentiation. For each addition chain $c_0, c_1, \ldots, c_\ell$, there is an algorithm that computes the c_ℓth power of its input with ℓ multiplications, by computing successively the c_1st power, the c_2nd power, etc.

For example, given m, and given an integer x between 0 and $m - 1$, one can compute successively

$$\begin{aligned}
x^2 \bmod m &= (x \cdot x) \bmod m, \\
x^3 \bmod m &= ((x^2 \bmod m) \cdot x) \bmod m, \\
x^5 \bmod m &= ((x^3 \bmod m) \cdot (x^2 \bmod m)) \bmod m, \\
x^7 \bmod m &= ((x^5 \bmod m) \cdot (x^2 \bmod m)) \bmod m, \\
x^{14} \bmod m &= ((x^7 \bmod m) \cdot (x^7 \bmod m)) \bmod m, \\
x^{28} \bmod m &= ((x^{14} \bmod m) \cdot (x^{14} \bmod m)) \bmod m, \\
x^{56} \bmod m &= ((x^{28} \bmod m) \cdot (x^{28} \bmod m)) \bmod m, \\
x^{63} \bmod m &= ((x^{56} \bmod m) \cdot (x^7 \bmod m)) \bmod m,
\end{aligned}$$

obtaining $x^{63} \bmod m$ with 8 multiplications modulo m.

Minimizing exponentiation time is not exactly the same problem as minimizing addition-chain length, for several reasons:

- Multiplication time is generally not constant. Multiplication of $x^e \bmod m$ by $x^f \bmod m$ may take less time if $e = f$, for example, or if $x^e \bmod m$ is small, or if $x^e \bmod m$ was read or written recently, or if $x^e \bmod m$ has participated in any previous multiplication.
- There may be several choices of (i, j) such that $c_k = c_i + c_j$. For example, the chain $1, 2, 3, 4, 7$ has $4 = 3 + 1$ and $4 = 2 + 2$. The choice sometimes affects the speed of exponentiation; the chain does not specify a choice.
- The fact that there *exists* an addition chain of length ℓ containing n does not mean that the programmer *knows* the addition chain. One often wants an algorithm that can compute an nth power, or several nth powers, given n as input. The time to find an addition chain is added to the time spent on multiplications, and must be minimized accordingly.
- Sometimes there are faster algorithms to compute $x^n \bmod m$. If m is prime and n is noticeably larger than m, for example, then one should first replace n with $1 + ((n - 1) \bmod (m - 1))$. Perhaps there are faster algorithms even when n is fairly small.

The model has nevertheless turned out to be useful. For example, Brauer proved in [9] that there is an addition chain of length $(1 + o(1)) \lg n$ containing n. "The proof yields an easy method for constructing the addition chains," Brauer wrote. Brauer's algorithm is explained in Section 3 of this paper; it is, for a wide range of m and n, nearly the fastest known algorithm to compute $x^n \bmod m$.

Finding shortest addition chains. Consider the set of $(n_1, \ldots, n_p, \ell)$ such that there is an addition chain of length ℓ containing $n_1, \ldots, n_p$. Downey, Leong, and Sethi in [14] proved that this set is NP-complete.

Many people incorrectly claim that Downey, Leong, and Sethi proved that the $p = 1$ subset is NP-complete. As far as I know, nobody has proven this. Perhaps there is a fast algorithm to find a minimum-length addition chain containing n.

Many people also misrepresent NP-completeness as an obstacle to finding chains that are *usually* the shortest possible.

Other semigroups. One can replace $\mathbf{Z}/m$ by any semigroup where multiplication is computable. If there is an addition chain of length ℓ containing n then there is an algorithm that computes nth powers in the semigroup with ℓ multiplications.

(There is, in fact, a **generic algorithm** that computes nth powers in every semigroup with ℓ multiplications. This means that the algorithm invokes an oracle ℓ times and that, for every semigroup H, the algorithm computes nth powers in H if the oracle computes products in H.)

Example: Given an element x of a finite extension ring of $\mathbf{Z}/m$, one can compute x^n with ℓ multiplications in the ring. Legendre in [27, pages 487–488] computed x^{105} in the ring $(\mathbf{Z}/211)[x]/(x^3 - 10x + 85)$ by computing successively $x^2, x^3, x^5, x^{10}, x^{20}, x^{25}, x^{40}, x^{80}, x^{105}$.

Another example: Given an integer x, one can compute nx with ℓ additions. For example, one can compute $5x$ with three additions. (Note that other algorithms are much faster when n and x are both large.) This is a popular way to multiply small integers on computers that have fast circuitry for addition but relatively slow circuitry for multiplication. "If we have no `MULTIPLY` instruction in our machine, we must simulate scalar multiplication by addition," Pippenger wrote in [34, page 258]. "This problem would be more realistic if we allowed negative coefficients, subtractions, and short shifts. These changes would not affect our analysis or results in any significant way."

Another example: Given a point x on an elliptic curve over a finite ring, one can compute the nth multiple of x on the curve with ℓ additions on the curve. This is another situation where subtractions are easy.

Last example: Given an integer x, one can compute x^n with ℓ multiplications. See [30], [19], [31], and [41] for optimization of addition chains in two models of this situation; note that multiplication time is not even close to constant.

3. Brauer's algorithm, 1939

Brauer in [9] published an algorithm that computes an nth power with $\lg n + (1 + o(1))(\lg n)/\lg\lg n$ multiplications.

The shortest addition chain containing n has length at least $\lg n$. Erdős proved in [17] that, for almost all n, the shortest addition chain containing n has length $\lg n + (1+o(1))(\lg n)/\lg\lg n$. Therefore Brauer's chain is always within a factor $1 + (1+o(1))/\lg\lg n$ of the shortest, and almost always within a factor $1+o(1)/\lg\lg n$.

Brauer's chain is actually a family of chains, parametrized by a positive integer k, and defined recursively as follows:

$$B_k(n) = \begin{cases} 1, 2, 3, \ldots, 2^k - 1 & \text{if } n < 2^k, \\ B_k(q), 2q, 4q, 8q, \ldots, 2^k q, n & \text{if } n \geq 2^k \text{ and } q = \lfloor n/2^k \rfloor. \end{cases}$$

1	=	1_2		
2	=	10_2		
3	=	11_2		
4	=	100_2		
5	=	101_2		
6	=	110_2		
7	=	111_2		
14	=	1110_2	=	$7 \cdot 2$
28	=	11100_2	=	$7 \cdot 2^2$
56	=	111000_2	=	$7 \cdot 2^3$
61	=	111101_2	=	$7 \cdot 2^3 + 5$
122	=	1111010_2	=	$7 \cdot 2^4 + 5 \cdot 2$
244	=	11110100_2	=	$7 \cdot 2^5 + 5 \cdot 2^2$
488	=	111101000_2	=	$7 \cdot 2^6 + 5 \cdot 2^3$
490	=	111101010_2	=	$7 \cdot 2^6 + 5 \cdot 2^3 + 2$
980	=	1111010100_2	=	$7 \cdot 2^7 + 5 \cdot 2^4 + 2 \cdot 2$
1960	=	11110101000_2	=	$7 \cdot 2^8 + 5 \cdot 2^5 + 2 \cdot 2^2$
3920	=	111101010000_2	=	$7 \cdot 2^9 + 5 \cdot 2^6 + 2 \cdot 2^3$
3926	=	111101010110_2	=	$7 \cdot 2^9 + 5 \cdot 2^6 + 2 \cdot 2^3 + 6$
7852	=	1111010101100_2	=	$7 \cdot 2^{10} + 5 \cdot 2^7 + 2 \cdot 2^4 + 6 \cdot 2$
15704	=	11110101011000_2	=	$7 \cdot 2^{11} + 5 \cdot 2^8 + 2 \cdot 2^5 + 6 \cdot 2^2$
31408	=	111101010110000_2	=	$7 \cdot 2^{12} + 5 \cdot 2^9 + 2 \cdot 2^6 + 6 \cdot 2^3$
31415	=	111101010110111_2	=	$7 \cdot 2^{12} + 5 \cdot 2^9 + 2 \cdot 2^6 + 6 \cdot 2^3 + 7$

FIGURE 1. Computing 31415 with 22 additions: Brauer's chain $B_3(31415)$.

In other words: Write n in radix 2^k as $2^{jk}c_j + \cdots + 2^{2k}c_2 + 2^k c_1 + c_0$, with $c_j \neq 0$. The chain is $1, 2, 3, \ldots, 2^k - 1$, then $2c_j, 4c_j, 8c_j, \ldots, 2^k c_j$, then $2^k c_j + c_{j-1}$, then $2^{k+1}c_j + 2c_{j-1}, \ldots, 2^{2k}c_j + 2^k c_{j-1}$, then $2^{2k}c_j + 2^k c_{j-1} + c_{j-2}$, and so on.

Brauer's chain has length $j(k+1) + 2^k - 2$ if $jk \leq \lg n < (j+1)k$. The length is minimized for k around $\lg\lg n - 2\lg\lg\lg n$.

Figure 1 shows Brauer's chain for $k = 3$ and $n = 31415$; Figure 2 shows Brauer's chain for $k = 3$ and $n = 27182$. Each chain has length 22. One can easily reduce this length to 20 in the first case and 19 in the second, as explained below. The minimum possible lengths of addition chains are 19 and 18 respectively.

If $2^{511} \leq n < 2^{512}$ then Brauer's chain has length 649 for $k = 4$, 642 for $k = 5$, and 657 for $k = 6$. For $k = 5$ one writes $n = 2^{510}c_{102} + \cdots + 2^5 c_1 + c_0$, where each of the 103 coefficients $c_0, c_1, \ldots, c_{102}$ is between 0 and 31 inclusive; Brauer's chain involves 30 preliminary steps, 510 doublings starting from c_{102}, and 102 additions of $c_{101}, c_{100}, \ldots, c_0$.

Brauer's algorithm is often called "the left-to-right 2^k-ary method," or simply "the 2^k-ary method." It is extremely popular. It is easy to implement; constructing the chain for n is a simple matter of inspecting the bits of n. It does not require much storage.

Minor improvement: start high. There is no reason for an addition chain to include a number twice. Knuth commented in [21] that one can omit $2q$ if $q < 2^{k-1}$, and $4q$ if $q < 2^{k-2}$, and so on. This saves $(-1 - \lfloor \lg n \rfloor) \bmod k$ steps, reducing the

1	=	1_2		
2	=	10_2		
3	=	11_2		
4	=	100_2		
5	=	101_2		
6	=	110_2		
7	=	111_2		
12	=	1100_2	=	$6 \cdot 2$
24	=	11000_2	=	$6 \cdot 2^2$
48	=	110000_2	=	$6 \cdot 2^3$
53	=	110101_2	=	$6 \cdot 2^3 + 5$
106	=	1101010_2	=	$6 \cdot 2^4 + 5 \cdot 2$
212	=	11010100_2	=	$6 \cdot 2^5 + 5 \cdot 2^2$
424	=	110101000_2	=	$6 \cdot 2^6 + 5 \cdot 2^3$
424	=	110101000_2	=	$6 \cdot 2^6 + 5 \cdot 2^3 + 0$
848	=	1101010000_2	=	$6 \cdot 2^7 + 5 \cdot 2^4 + 0 \cdot 2$
1696	=	11010100000_2	=	$6 \cdot 2^8 + 5 \cdot 2^5 + 0 \cdot 2^2$
3392	=	110101000000_2	=	$6 \cdot 2^9 + 5 \cdot 2^6 + 0 \cdot 2^3$
3397	=	110101000101_2	=	$6 \cdot 2^9 + 5 \cdot 2^6 + 0 \cdot 2^3 + 5$
6794	=	1101010001010_2	=	$6 \cdot 2^{10} + 5 \cdot 2^7 + 0 \cdot 2^4 + 5 \cdot 2$
13588	=	11010100010100_2	=	$6 \cdot 2^{11} + 5 \cdot 2^8 + 0 \cdot 2^5 + 5 \cdot 2^2$
27176	=	110101000101000_2	=	$6 \cdot 2^{12} + 5 \cdot 2^9 + 0 \cdot 2^6 + 5 \cdot 2^3$
27182	=	110101000101110_2	=	$6 \cdot 2^{12} + 5 \cdot 2^9 + 0 \cdot 2^6 + 5 \cdot 2^3 + 6$

FIGURE 2. Computing 27182 with 22 additions: Brauer's chain $B_3(27182)$.

chain length to $\lfloor \lg n \rfloor + j + 2^k - k - 1$: for example, 639 if $2^{511} \le n < 2^{512}$ and $k = 5$.

Minor improvement: eliminate zero digits. Knuth also commented that one can compress two steps $2^k q, n$ to a single step when $n = 2^k q$. For example, in Figure 2, one can omit the second 424. If $2^{511} \le n < 2^{512}$ and $k = 5$ then there are typically 3 or 4 zeros among $c_0, c_1, \ldots, c_{101}$, saving 3 or 4 steps.

Minor improvement: eliminate even digits. Thurber pointed out in [39] that, if $k \ge 3$ and $n \ge 2^k$, one can eliminate the numbers $4, 6, 8, 10, \ldots, 2^k - 2$ from Brauer's chain, reducing the chain length to $2^{k-1} + (k+1)j$. The idea is to change "double, then add r" into "add $r/2$, then double" if r is even; and to compute $2c_j$ as $(c_j - 1) + (c_j + 1)$ if c_j is even.

For example, in Figure 1, one can omit 4 and 6, changing "double, then add 6" into "add 3, then double" later in the chain. The same works in Figure 2: one can construct 12 as $5 + 7$ without 6. Thurber's improvement saves 14 steps for $k = 5$, 30 steps for $k = 6$, etc.

Belaga independently made the same observation in [4, page 7] a few years later. Belaga's proof applies only if c_j is odd.

Minor improvement: vary the exponents. Thurber also pointed out that one can usually reduce the number of nonzero coefficients in the expansion $n = 2^{jk} c_j + \cdots + 2^{2k} c_2 + 2^k c_1 + c_0$ by allowing some flexibility in the exponents $0, k, 2k, \ldots, jk$.

1	=	1_2		
2	=	10_2		
3	=	11_2		
4	=	100_2		
5	=	101_2		
6	=	110_2		
7	=	111_2		
6	=	110_2	=	$3 \cdot 2$
12	=	1100_2	=	$3 \cdot 2^2$
24	=	11000_2	=	$3 \cdot 2^3$
25	=	11001_2	=	$3 \cdot 2^3 + 1$
50	=	110010_2	=	$3 \cdot 2^4 + 2$
100	=	1100100_2	=	$3 \cdot 2^5 + 2^2$
200	=	11001000_2	=	$3 \cdot 2^6 + 2^3$
204	=	11001100_2	=	$3 \cdot 2^6 + 2^3 + 4$
408	=	110011000_2	=	$3 \cdot 2^7 + 2^4 + 4 \cdot 2$
816	=	1100110000_2	=	$3 \cdot 2^8 + 2^5 + 4 \cdot 2^2$
1632	=	11001100000_2	=	$3 \cdot 2^9 + 2^6 + 4 \cdot 2^3$
1639	=	11001100111_2	=	$3 \cdot 2^9 + 2^6 + 4 \cdot 2^3 + 7$
3278	=	110011001110_2	=	$3 \cdot 2^{10} + 2^7 + 4 \cdot 2^4 + 7 \cdot 2$
6556	=	1100110011100_2	=	$3 \cdot 2^{11} + 2^8 + 4 \cdot 2^5 + 7 \cdot 2^2$
13112	=	11001100111000_2	=	$3 \cdot 2^{12} + 2^9 + 4 \cdot 2^6 + 7 \cdot 2^3$
13113	=	11001100111001_2	=	$3 \cdot 2^{12} + 2^9 + 4 \cdot 2^6 + 7 \cdot 2^3 + 1$

FIGURE 3. Computing 13113 with 22 additions: Brauer's chain $B_3(13113)$.

1	=	1_2		
3	=	11_2		
5	=	101_2		
7	=	111_2		
12	=	1100_2	=	$6 \cdot 2$
24	=	11000_2	=	$6 \cdot 2^2$
48	=	110000_2	=	$6 \cdot 2^3$
51	=	110011_2	=	$6 \cdot 2^3 + 3$
102	=	1100110_2	=	$6 \cdot 2^4 + 3 \cdot 2$
204	=	11001100_2	=	$6 \cdot 2^5 + 3 \cdot 2^2$
408	=	110011000_2	=	$6 \cdot 2^6 + 3 \cdot 2^3$
816	=	1100110000_2	=	$6 \cdot 2^7 + 3 \cdot 2^4$
1632	=	11001100000_2	=	$6 \cdot 2^8 + 3 \cdot 2^5$
1639	=	11001100111_2	=	$6 \cdot 2^8 + 3 \cdot 2^5 + 7$
3278	=	110011001110_2	=	$6 \cdot 2^9 + 3 \cdot 2^6 + 7 \cdot 2$
6556	=	1100110011100_2	=	$6 \cdot 2^{10} + 3 \cdot 2^7 + 7 \cdot 2^2$
13112	=	11001100111000_2	=	$6 \cdot 2^{11} + 3 \cdot 2^8 + 7 \cdot 2^3$
13113	=	11001100111001_2	=	$6 \cdot 2^{11} + 3 \cdot 2^8 + 7 \cdot 2^3 + 1$

FIGURE 4. Computing 13113 with 17 additions.

For example, $3\cdot 2^{12}+2^9+4\cdot 2^6+7\cdot 2^3+1$ can be written as $6\cdot 2^{11}+3\cdot 2^8+7\cdot 2^3+1$. Compare Figure 3 to Figure 4. The nonzero coefficients here are often visualized as "windows" through which one can see all the nonzero bits of n:

$$\begin{aligned} 3\cdot 2^{12}+2^9+4\cdot 2^6+7\cdot 2^3+1 &= \boxed{\ \ 1\ 1}\boxed{0\ 0\ 1}\boxed{1\ 0\ 0}\boxed{1\ 1\ 1}\boxed{0\ 0\ 1} \\ 6\cdot 2^{11}+3\cdot 2^8+7\cdot 2^3+1 &= \ \ \boxed{1\ 1\ 0}\ 0\ \boxed{1\ 1}\ 0\ 0\ \boxed{1\ 1\ 1}\ 0\ 0\ \boxed{1} \end{aligned}$$

The idea of varying the exponents is often called "sliding windows."

One popular chain, combining all of the above improvements, is the chain defined recursively for $k\ge 2$ and $n\ge 2^k$ as follows:

$$T_k(n)=\begin{cases} 1,2,3,5,7,\ldots,2^k-1,n & \text{if } n<2^{k+1} \text{ and } n \text{ is even,}\\ T_k(n/2),n & \text{if } n\ge 2^{k+1} \text{ and } n \text{ is even,}\\ T_k(n-(n \bmod 2^{\lceil \lg n\rceil -k})),n & \text{if } n<2^{2k} \text{ and } n \text{ is odd,}\\ T_k(n-(n \bmod 2^k)),n & \text{if } n\ge 2^{2k} \text{ and } n \text{ is odd.}\end{cases}$$

The leftmost window always has k bits; subsequent windows start with a 1, end with a 1, and have at most k bits. If $2^{511}\le n<2^{512}$ and $k=5$ then the chain length is typically about 608.

Minor improvement: use negative digits. Coefficients $-1,-3,\ldots,-(2^k-1)$ are just as good as $1,3,\ldots,2^k-1$ for semigroups where division is as easy as multiplication, such as the group of points on an elliptic curve. One can round n to the nearest multiple of 2^{k+1} instead of rounding it down to the nearest multiple of 2^k. This typically saves about 12 multiplications if $2^{511}\le n<2^{512}$ and $k=5$.

Minor improvement: limit the precomputation. There is no reason for an addition chain to include a number that will not be used. Knuth commented that "often" one can omit steps from the initial sequence $1,2,3,\ldots,2^k-1$. This is not true if $2^{511}\le n<2^{512}$ and $k=5$, for example, but it is true when k is chosen somewhat larger.

The initial sequence $1,2,3,\ldots,2^k-1$ can be replaced by any addition chain that contains $c_0,c_1,\ldots,c_j$. Thurber pointed this out explicitly in [39, page 912]. Thurber gave the example $516723=63\cdot 2^{16}+39\cdot 2^4+3$, with addition chain starting $1,2,3,6,9,15,24,39,63$ and continuing in the obvious way.

One could use Yao's algorithm or Pippenger's algorithm to find an addition chain containing $c_0,c_1,\ldots,c_j$. Or one can spend more time searching for a better chain; see, e.g., [7, pages 405–406].

Multiple inputs. Straus in [38] generalized Brauer's algorithm to compute a product $x_1^{n_1}x_2^{n_2}\ldots x_p^{n_p}$; in other words, to obtain the vector $(n_1,n_2,\ldots,n_p)$ by additions starting from the unit vectors.

Straus's algorithm uses $j(k+1)+2^{pk}-p-1$ additions if $jk\le \lg B\le (j+1)k$, where B is the maximum of $n_1,n_2,\ldots,n_p$. Write $(n_1,n_2,\ldots,n_p)$ in radix 2^k; each coefficient is a vector $(r_1,r_2,\ldots,r_p)$ with $r_1,r_2,\ldots,r_p\in\{0,1,\ldots,2^k-1\}$. The algorithm obtains all of those vectors with $2^{pk}-p-1$ additions, and then obtains $(n_1,n_2,\ldots,n_p)$ with $(k+1)j$ additions.

If p is fixed and $B\to\infty$ then Straus's algorithm uses $\lg B+(p+o(1))(\lg B)/\lg\lg B$ additions with the optimal choice of k.

$$
\begin{array}{rcl}
(1,0) & & \\
(0,1) & & \\
(1,1) & = & (1,0)+(0,1) \\
(2,2) & = & (1,1)+(1,1) \\
(4,4) & = & (2,2)+(2,2) \\
(5,5) & = & (4,4)+(1,1) \\
(6,5) & = & (5,5)+(1,0) \\
(7,6) & = & (6,5)+(1,1) \\
(14,12) & = & (7,6)2 \\
(28,24) & = & (7,6)2^2 \\
(56,48) & = & (7,6)2^3 \\
(61,53) & = & (7,6)2^3+(5,5) \\
(122,106) & = & (7,6)2^4+(5,5)2 \\
(244,212) & = & (7,6)2^5+(5,5)2^2 \\
(245,212) & = & (7,6)2^5+(5,5)2^2+(1,0) \\
(490,424) & = & (7,6)2^6+(5,5)2^3+(1,0)2 \\
(980,848) & = & (7,6)2^7+(5,5)2^4+(1,0)2^2 \\
(1960,1696) & = & (7,6)2^8+(5,5)2^5+(1,0)2^3 \\
(3920,3392) & = & (7,6)2^9+(5,5)2^6+(1,0)2^4 \\
(3926,3397) & = & (7,6)2^9+(5,5)2^6+(1,0)2^4+(6,5) \\
(7852,6794) & = & (7,6)2^{10}+(5,5)2^7+(1,0)2^5+(6,5)2 \\
(15704,13588) & = & (7,6)2^{11}+(5,5)2^8+(1,0)2^6+(6,5)2^2 \\
(31408,27176) & = & (7,6)2^{12}+(5,5)2^9+(1,0)2^7+(6,5)2^3 \\
(31415,27182) & = & (7,6)2^{12}+(5,5)2^9+(1,0)2^7+(6,5)2^3+(7,6)
\end{array}
$$

FIGURE 5. Computing the vector (31415, 27182) with 22 additions.

The minor improvements described above can be generalized to this situation. Figure 5, for example, shows how to compute $x_1^{31415}x_2^{27182}$, given x_1 and x_2, with just 22 multiplications.

One variant of Straus's algorithm is to build each $(r_1, r_2, \dots, r_p)$ upon demand as a sum of $(r_1, 0, \dots, 0)$ and $(0, r_2, \dots, 0)$ and so on. This algorithm is typically a little slower than Straus's algorithm; it uses $j(k+p) + (2^k - 1)p - 1$ additions.

Setting the record straight. Thurber's improvements in [39] were reinvented and republished two decades later in [26] and [6].

ElGamal in [16] presented the special case $(p, k) = (2, 1)$ of Straus's algorithm as "Shamir's trick." Many subsequent papers give credit to Shamir, rather than Straus, for the particular algorithm, and for the general observation that computing $x_1^{n_1}x_2^{n_2}$ is faster than computing $x_1^{n_1}, x_2^{n_2}$. I see no reason that Shamir should receive any credit here.

4. Yao's algorithm, 1976

Yao in [40] published an algorithm that, like Brauer's algorithm, computes an nth power with $\lg n + (1 + o(1))(\lg n)/\lg\lg n$ multiplications.

1	=	1_2					
2	=	10_2					
4	=	100_2					
8	=	1000_2					
16	=	10000_2					
32	=	100000_2					
64	=	1000000_2	520	=	1000001000_2	=	$d(5)$
128	=	10000000_2	1040	=	10000010000_2	=	$2d(5)$
256	=	100000000_2	2080	=	100000100000_2	=	$4d(5)$
512	=	1000000000_2	2600	=	101000101000_2	=	$5d(5)$
1024	=	10000000000_2	4097	=	1000000000001_2	=	$d(6)$
2048	=	100000000000_2	8194	=	10000000000010_2	=	$2d(6)$
4096	=	1000000000000_2	16388	=	100000000000100_2	=	$4d(6)$
8192	=	10000000000000_2	24582	=	110000000000110_2	=	$6d(6)$
16384	=	100000000000000_2	27182	=	110101000101110_2		

FIGURE 6. Computing 27182 with 23 additions: Yao's chain.

Select a positive integer k. Write n in radix 2^k as $2^{jk}c_j + \cdots + 2^{2k}c_2 + 2^k c_1 + c_0$, with $c_j \neq 0$, exactly as in Section 3. Define $d(z)$ as the sum of 2^{ik} over all i such that $c_i = z$.

Yao's chain begins with $1, 2, 4, 8, \ldots, 2^{\lfloor \lg n \rfloor}$; adds various 2^{ik} together to obtain $d(z)$ for each $z \in \{1, 2, 3, \ldots, 2^k - 1\}$ such that $d(z)$ is nonzero; then obtains $zd(z)$ for each z; and finally obtains $n = d(1) + 2d(2) + 3d(3) + \cdots + (2^k - 1)d(2^k - 1)$.

Figure 6 shows Yao's chain for 27182. The chain has length 23.

Minor improvements. One can start Yao's chain with $1, 2, 4, 8, \ldots, 2^{jk}$, saving $\lfloor \lg n \rfloor - jk$ steps. The rest of the chain uses only $1, 2^k, 2^{2k}, \ldots, 2^{jk}$.

It is usually best to compute $n = \sum_z zd(z)$ as the sum of the terms $d(2^k - 1)$, $d(2^k - 1) + d(2^k - 2)$, and so on through $d(2^k - 1) + d(2^k - 2) + \cdots + d(2) + d(1)$. These terms can in turn be computed with at most $2^k - 2$ successive additions. The number of additions depends on which $d(z)$'s are 0.

Yao's algorithm, with these two basic improvements, is exactly the algorithm shown in [22, answer to exercise 4.6.3–9]. The improved chain has length at most $j(k+1) + 2^k - 2$, just like Brauer's chain. Each of the improvements to Brauer's chain explained in Section 3 can be matched by a further improvement to Yao's chain. This is not a coincidence—see Section 5.

Multiple outputs. Yao's chain is mostly independent of n. Computing x^n takes just $j + 2^k - 2$ multiplications, after the initial computation of $x^{2^k}, x^{2^{2k}}, \ldots, x^{2^{jk}}$. (The formula $j + 2^k - 2$ assumes the two basic improvements. Similar comments apply without the improvements, but the formula becomes more complicated.)

Obviously one can use Yao's algorithm to compute many powers $x^{n_1}, x^{n_2}, \ldots, x^{n_p}$ quickly. The union of Yao's chains for $n_1, n_2, \ldots, n_k$ is a chain of length $j(k+p) + (2^k - 2)p$ containing $n_1, n_2, \ldots, n_p$, provided that $\lg n_1, \lg n_2, \ldots, \lg n_p$ are smaller than $(j+1)k$. Minor improvements are again possible.

Yao concluded in [40, page 103] that there is an addition chain containing $n_1, n_2, \ldots, n_p$ of length at most $\lg B + (p + o(1))(\lg B)/\lg\lg B$ if p is fixed, $B \to \infty$, and $n_1, n_2, \ldots, n_p < B$.

Setting the record straight. Knuth failed to give Yao credit for the algorithm in [22, answer to exercise 4.6.3–9]. Knuth obtained the algorithm in a different way, by feeding Brauer's algorithm to the machinery explained in Section 5, but this discovery was not independent of Yao's paper; Knuth's knowledge of the machinery can be traced back to Yao's paper. Knuth did not notice that the algorithm was mostly independent of the exponent, or that it was (aside from minor improvements) the same as Yao's algorithm. Knuth did, however, present Yao's algorithm with proper credit in [22, answer to exercise 4.6.3–32].

Many years later, unaware of both [22, answer to exercise 4.6.3–9] and [22, answer to exercise 4.6.3–32], Brickell, Gordon, McCurley, and Wilson in [10] announced that one could compute a power with a chain mostly independent of the exponent. The Brickell-Gordon-McCurley-Wilson algorithm is identical to Yao's algorithm for multiple outputs with the two basic improvements.

I have seen dozens of papers assigning credit to [10]. In every case, credit should instead have been assigned to Yao in [40].

Adding injury to insult, Sandia National Laboratories obtained United States Patent 5299262 on behalf of Brickell, Gordon, and McCurley, and then sold the patent to RSA Data Security, Inc. The patent was filed in August 1992; it will expire in August 2012. The patent is clearly invalid, and I will be happy to testify to that effect.

One can reasonably give credit to Brickell, Gordon, McCurley, and Wilson for the two basic improvements on Yao's algorithm for multiple outputs. Yao's paper did not have the two basic improvements. Knuth's exercise had the two basic improvements, but was not recognized as a variant of Yao's algorithm.

5. Transposition

This section presents a matrix formulation of addition chains, and then explains what happens when the matrix is transposed. Special case: The transpose of a Brauer-Straus-type algorithm is a Yao-type algorithm, and vice versa.

It is difficult to assign credit for algorithm transposition. The long and messy history is summarized at the end of this section. The fact that transposition is interesting even for a single power was pointed out by Knuth and Papadimitriou in [25], and highlighted by Knuth in [22, pages 460–462].

Matrices. Consider once again the addition chain $(c_0, c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8) = (1, 2, 3, 5, 7, 14, 28, 56, 63)$, proven to be an addition chain by the formulas $c_0 = 1$, $c_1 = c_0 + c_0$, $c_2 = c_1 + c_0$, $c_3 = c_2 + c_1$, $c_4 = c_3 + c_1$, $c_5 = c_4 + c_4$, $c_6 = c_5 + c_5$, $c_7 = c_6 + c_6$, $c_8 = c_7 + c_4$. Let M be the matrix of coefficients of $c_0, c_1, \ldots, c_8$ in these formulas for $c_0, c_1, \ldots, c_8$:

$$M = \begin{pmatrix} 0&0&0&0&0&0&0&0&0\\ 2&0&0&0&0&0&0&0&0\\ 1&1&0&0&0&0&0&0&0\\ 0&1&1&0&0&0&0&0&0\\ 0&1&0&1&0&0&0&0&0\\ 0&0&0&0&2&0&0&0&0\\ 0&0&0&0&0&2&0&0&0\\ 0&0&0&0&0&0&2&0&0\\ 0&0&0&0&1&0&0&1&0 \end{pmatrix} \qquad \begin{array}{l} \\ \text{since } c_1 = 2c_0, \\ \text{since } c_2 = 1c_0 + 1c_1, \\ \text{since } c_3 = 1c_1 + 1c_2, \\ \text{since } c_4 = 1c_1 + 1c_3, \\ \text{since } c_5 = 2c_4, \\ \text{since } c_6 = 2c_5, \\ \text{since } c_7 = 2c_6, \\ \text{since } c_8 = 1c_4 + 1c_7. \end{array}$$

An easy calculation shows that

$$\frac{1}{1-M} = \begin{pmatrix} 1&0&0&0&0&0&0&0&0\\ 2&1&0&0&0&0&0&0&0\\ 3&1&1&0&0&0&0&0&0\\ 5&2&1&1&0&0&0&0&0\\ 7&3&1&1&1&0&0&0&0\\ 14&6&2&2&2&1&0&0&0\\ 28&12&4&4&4&2&1&0&0\\ 56&24&8&8&8&4&2&1&0\\ 63&27&9&9&9&4&2&1&1 \end{pmatrix}.$$

The first column of $1/(1-M)$ is the original chain $1, 2, 3, 5, 7, 14, 28, 56, 63$. Note also that the last row of $1/(1-M)$ shares many numbers with the Yao-type chain $1, 2, 4, 8, 9, 18, 27, 54, 63$.

More generally, let M be a lower-triangular nonnegative integer matrix with zero diagonal. Typically each entry of M will be in $\{0, 1, 2, 3\}$, although this is not required. Define $P = 1/(1-M) = 1 + M + M^2 + M^3 + \cdots$.

The standard method to compute P, given M, is **substitution**: computing P one row at a time from the formula $P = 1 + MP$. This method exploits the fact that M is lower-triangular with zero diagonal, i.e., that $M_{ki} = 0$ for $i \geq k$. The kth row P_k is $e_k + \sum_i M_{ki}P_i = e_k + \sum_{i<k} M_{ki}P_i$, where e_k is the kth unit vector; so one can compute P_k from $e_k, P_1, \ldots, P_{k-1}$ with $\sum_i M_{ki}$ vector additions, by adding each P_i to the current sum M_{ki} times.

The total number of vector additions to compute all the rows $P_1, P_2, \ldots$ of P, starting from the unit vectors, is the sum $\sum_{k,i} M_{ki}$ of all the entries of M.

Often one wants only a few entries of these vectors. One can compress each vector by discarding v columns, and skip the additions of the corresponding unit vectors. It then takes $(\sum_{k,i} M_{ki}) - v$ vector additions to compute the compressed rows of P, starting from the unit vectors and the zero vector. Every addition-chain algorithm discussed in this paper can be expressed in this form.

The same comments apply to an *upper-triangular* nonnegative integer matrix M with zero diagonal, except that the rows of P are computed from bottom to top rather than top to bottom.

Consider, as another example, the matrices

$$M = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 & 2 & 0 \end{pmatrix} \quad \text{and} \quad P = \frac{1}{1-M} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 & 0 & 0 \\ 7 & 2 & 1 & 0 & 0 & 0 \\ 14 & 4 & 2 & 1 & 0 & 0 \\ 28 & 8 & 4 & 2 & 1 & 0 \\ 63 & 18 & 9 & 4 & 2 & 1 \end{pmatrix}.$$

The sum of the entries of M is 13, so computing $P_1, P_2, P_3, P_4, P_5, P_6$ from the unit vectors as explained above takes 13 vector additions. For example, the last row of M is $(0,0,1,0,2,0)$, so $P_6 = e_6 + P_3 + P_5 + P_5 = (0,0,0,0,0,1) + (7,2,1,0,0,0) + (28,8,4,2,1,0) + (28,8,4,2,1,0)$; computing P_6 from e_6, P_3, P_5 takes 3 additions.

Computing the first, third, and fourth entries of each vector, starting from $(1,0,0)$ and $(0,1,0)$ and $(0,0,1)$ and $(0,0,0)$, takes only $13 - 3 = 10$ additions:

$$\begin{aligned} (3,0,0) &= (1,0,0) + (1,0,0) + (1,0,0), \\ (7,1,0) &= (0,1,0) + (3,0,0) + (3,0,0) + (1,0,0), \\ (14,2,1) &= (0,0,1) + (7,1,0) + (7,1,0), \\ (28,4,2) &= (14,2,1) + (14,2,1), \\ (63,9,4) &= (7,1,0) + (28,4,2) + (28,4,2). \end{aligned}$$

Computing the first entry of each vector, starting from 1 and 0, takes only $13 - 5 = 8$ additions: $3 = 1 + 1 + 1$, $7 = 1 + 3 + 3$, $14 = 7 + 7$, $28 = 14 + 14$, $63 = 7 + 28 + 28$.

Transposition. One can use the formula $P = 1 + PM$ to compute the *column* vectors of P one at a time, right to left, starting from the unit vectors. The number of *column* additions here is the sum of all the entries of M.

For example, one can compute the column vector $(1,3,7,14,28,63)$ from the unit vectors with 13 vector additions.

This column-by-column algorithm is the **transpose** of the row-by-row algorithm. It is the row-by-row algorithm applied to the transpose M^* of M; note that $P^* = 1/(1 - M^*)$.

Transposition is often phrased in the language of graphs. The matrix M is viewed as a digraph with vertices $1, 2, 3, \ldots$ and with M_{ki} edges from vertex i to vertex k, or equivalently one edge of weight M_{ki} from vertex i to vertex k. The matrix $P = 1/(1 - M)$ counts paths in the digraph: the number of paths from vertex i to vertex k in the digraph is exactly P_{ki}. The transpose of M is the same digraph, except that all edges are reversed.

Examples of transposition. Recall from Section 4 that, given x_1, x_2, x_3, x_4, x_5, one can compute $x_5^5 x_4^4 x_3^3 x_2^2 x_1$ with 8 multiplications: $(1,1,0,0,0) = (1,0,0,0,0) + (0,1,0,0,0)$, $(1,1,1,0,0) = (0,0,1,0,0) + (1,1,0,0,0)$, $(1,1,1,1,0) = (0,0,0,1,0) + (1,1,1,0,0)$, $(1,1,1,1,1) = (0,0,0,0,1) + (1,1,1,1,0)$, $(5,4,3,2,1) = (1,0,0,0,0) + (1,1,0,0,0) + (1,1,1,0,0) + (1,1,1,1,0) + (1,1,1,1,1)$. This computation is the transpose of the simple computation $2 = 1 + 1$, $3 = 1 + 2$, $4 = 1 + 3$, $5 = 1 + 4$.

More generally: The improved version of Yao's x^n algorithm shown in [22, answer to exercise 4.6.3–9] is the transpose of Brauer's x^n algorithm (without repeated and unused numbers). The further improvement from [22, answer to exercise 4.6.3–9] to [23, answer to exercise 4.6.3–9] is the transpose of Thurber's elimination of even digits in Brauer's algorithm.

Yao's original x^n algorithm is the transpose of a clumsier version of Brauer's algorithm, in which $x^2, x^3, x^4, \ldots, x^{2^k-1}$ are computed separately instead of by successive multiplications.

Yao's $x^{n_1}, x^{n_2}, \ldots, x^{n_p}$ algorithm is almost, though not exactly, the transpose of Straus's $x_1^{n_1} x_2^{n_2} \ldots x_p^{n_p}$ algorithm. It is actually the transpose of the slightly slower variant of Straus's algorithm described at the end of Section 3, plus the clumsiness mentioned above.

The reader may enjoy transposing Straus's algorithm: in particular, transposing Figure 5 to find an addition chain of length 21 containing 27182 and 31415.

Setting the record straight. Lupanov in [29] pointed out a less general form of transposition, namely transposition for Boolean matrices. Here one considers the positions, but not the values, of nonzero entries in M and $1/(1-M)$. This is second-hand information; I'm still looking for an English translation of Lupanov's article.

Fiduccia in [18] considered general linear computations, allowing additions and scalar multiplications. Fiduccia observed that the transpose of any computation of a matrix is a computation of the transpose of the matrix. This is also second-hand information; I'm still looking for [18] and Fiduccia's 1973 thesis at Brown.

Pippenger in [34, page 259] observed (in graph-theoretic language) that sequences of additions specified by M are computations of submatrices of $1/(1-M)$. He also observed that transposing M would transpose the submatrices.

I should comment on the different proof strategies here. I also have to check [2].

Transposition was subsequently reinvented and republished as the centerpiece of three papers, all of which cited [34]:

- Olivos in [33] proved that the minimum number of vector additions needed to compute $(n_1, n_2, \ldots, n_p)$, given the unit vectors, is $p-1$ more than the minimum length of an addition chain containing $n_1, n_2, \ldots, n_p$, if none of $n_1, n_2, \ldots, n_p$ are 0. Olivos's proof laboriously constructs transpositions without the help of graphs or matrices.
- Knuth and Papadimitriou in [25] proved that, more generally, the minimum number of vector additions needed to compute the column vectors of a $p \times q$ matrix, given the unit vectors, is $p-q$ more than the number of vector additions needed to compute the row vectors of the matrix, if none of the row and column vectors are 0. Knuth and Papadimitriou said that they were using "a graph-theoretic formulation of the problem—first employed by Pippenger."
- Kaminski, Kirkpatrick, and Bshouty in [20] published transposition in the same level of generality as Fiduccia, calling it a "modest generalization" of Lupanov's results and Pippenger's results.

My feeling at this point is that Fiduccia deserves much more credit, but obviously I have to find Fiduccia's papers before rendering final judgment.

6. Pippenger's multiple-product algorithm, 1976

Pippenger's multiple-product algorithm in [34] computes products of specified subsequences of a sequence. In other words, it computes products of powers, where all the exponents are in $\{0, 1\}$. In other words, it obtains specified vectors over $\{0, 1\}$

by additions starting from the unit vectors. In other words, it obtains specified sets by disjoint unions starting from singletons.

Pippenger's exponentiation algorithm uses the multiple-product algorithm as a subroutine, as explained in Section 7.

Pippenger's multiple-product algorithm has two inputs: a sequence $x_1, \ldots, x_p$, and a sequence of nonempty subsets $S_1, S_2, \ldots, S_q$ of $\{1, \ldots, p\}$. The output is the corresponding sequence of products $X(S_1), X(S_2), \ldots, X(S_q)$, where $X(S) = \prod_{i \in S} x_i$. For example, if the inputs are x_1, x_2, x_3, x_4 and $\{1,3\}, \{1,2,4\}, \{2,3,4\}$, then the output is $x_1x_3, x_1x_2x_4, x_2x_3x_4$.

Option 1. In some cases, Pippenger's algorithm computes each desired product separately. The total number of multiplications here is $w-q$, where w is the **weight** of $S_1, S_2, \ldots, S_q$: the sum $\#S_1 + \#S_2 + \cdots + \#S_q$. This is a good choice when w is small.

Option 2. In some cases, Pippenger's algorithm **partitions the input**. Figure 7 summarizes input partitioning.

Input partitioning is a generalization of Kronrod's Boolean matrix-multiplication algorithm in [3]. It works as follows. Choose an integer $c \geq 2$; typically c will be logarithmic in pq. Partition $\{1, 2, 3, \ldots, p\}$ into $\lceil p/c \rceil$ parts, each part of size c or smaller: $P_0 = \{1, 2, \ldots, c\}$ and $P_1 = \{c+1, c+2, \ldots, 2c\}$ and so on.

Observe that $X(S) = X(P_0 \cap S)X(P_1 \cap S) \cdots$. For each j, compute $X(T)$ for all nonempty subsets T of P_j, from smallest to largest; this takes at most $\lceil p/c \rceil (2^c - c - 1)$ multiplications and produces at most $\lceil p/c \rceil (2^c - 1)$ values $X(T)$. Now use Pippenger's algorithm recursively to compute each desired $X(S)$ as the product of all relevant $X(T)$.

Consider, for example, inputs x_1, x_2, x_3, x_4 and $13, 124, 234$. Choose $c = 2$. Compute $X(1) = x_1$, $X(2) = x_2$, $X(12) = x_1x_2$, $X(3) = x_3$, $X(4) = x_4$, and $X(34) = x_3x_4$. Then use Pippenger's algorithm recursively to compute $X(1)X(3)$, $X(12)X(4), X(2)X(34)$.

Another example: Consider inputs x_1, x_2, x_3, x_4 and $134, 12, 1234$. Then the recursive step is to compute $X(1)X(34), X(12), X(12)X(34)$. The total number of multiplications here is 4.

Larger example: Consider inputs $x_1, x_2, \ldots, x_8$ and $134567, 1235, 145678, 234578, 124568, 123478, 1234568, 1357, 2348, 13567$. Choose $c = 3$. Compute $X(1) = x_1$, $X(2) = x_2$, $X(3) = x_3$, $X(12) = x_1x_2$, $X(13) = x_1x_3$, $X(23) = x_2x_3$, $X(123) = X(12)x_3$, $X(4) = x_4$, $X(5) = x_5$, $X(6) = x_6$, $X(45) = x_4x_5$, $X(46) = x_4x_6$,

$$
\begin{array}{c}
p \text{ inputs: } x_1, x_2, \ldots, x_p \\
\Big\downarrow \text{at most } \lceil p/c \rceil (2^c - c - 1) \text{ multiplications} \\
\text{at most } \lceil p/c \rceil (2^c - 1) \text{ intermediate } X(T)\text{'s} \\
\Big\downarrow \text{recursively; weight at most } \lceil p/c \rceil q \\
q \text{ outputs}
\end{array}
$$

FIGURE 7. Input partitioning.

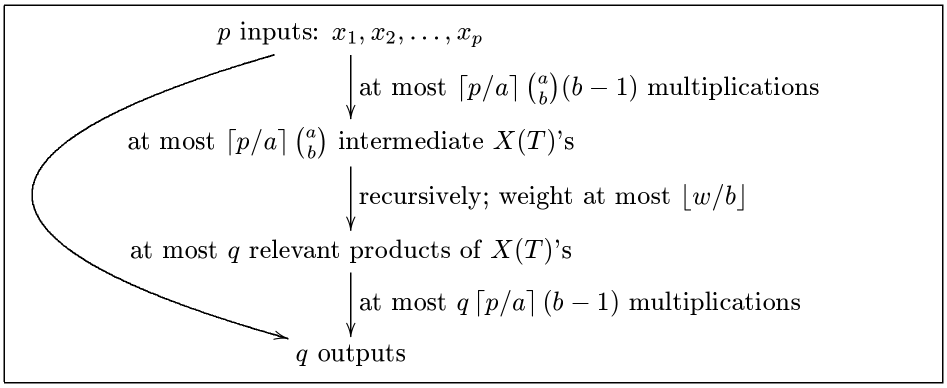

FIGURE 8. Input clumping.

$X(56) = x_5x_6$, $X(456) = X(45)x_6$, $X(7) = x_7$, $X(8) = x_8$, $X(78) = x_7x_8$. Then use Pippenger's algorithm recursively to compute $X(13)X(456)X(7)$ and $X(123)X(5)$ and so on.

Option 3. In some cases, Pippenger's algorithm **clumps the input**. Figure 8 summarizes input clumping.

Input clumping works as follows. Choose an integer $b \geq 2$ and an integer $a \geq b$. Typically b will be 2, while a will be fairly large, only a polylogarithmic factor smaller than p. Partition $\{1, 2, 3, \ldots, p\}$ into $\lceil p/a \rceil$ parts, each part of size a or smaller: $P_0 = \{1, 2, \ldots, a\}$ and $P_1 = \{a+1, a+2, \ldots, 2a\}$ and so on.

For each j, compute $X(T)$ for every size-b subset T of P_j, by multiplying the relevant x's. There are at most $\lceil p/a \rceil \binom{a}{b}$ such $T's$, and each one involves $b-1$ multiplications. (There is some room for improvement here when $b \geq 3$.)

Choose a decomposition of each input set S as a disjoint union of as many T's as possible, plus a few overflow elements—at most $b-1$ overflow elements in each part, totaling at most $\lceil p/a \rceil (b-1)$ overflow elements.

Use Pippenger's algorithm recursively to compute all the relevant products of $X(T)$'s; the weight here is at most $\lfloor w/b \rfloor$, because at most $\#S/b$ disjoint size-b sets T can fit inside S. Then multiply these products by the overflow x's to compute the desired products $X(S)$; there are at most $q \lceil p/a \rceil (b-1)$ overflow multiplications.

Example: Consider again $134567, 1235, 145678, 234578, 124568, 123478, 1234568, 1357, 2348, 13567$. Choose $b = 2$ and $a = 4$. Compute $X(12) = x_1x_2$, $X(13) = x_1x_3$, $X(14) = x_1x_4$, $X(23) = x_2x_3$, $X(24) = x_2x_4$, $X(34) = x_3x_4$; similarly compute $X(56), X(57), X(58), X(67), X(68), X(78)$. Use Pippenger's algorithm recursively to compute $X(13)X(56)$, $X(12)$, $X(14)X(56)X(78)$, etc. The desired outputs are then $X(13)X(56)x_4$, $X(12)x_3x_5$, $X(14)X(56)X(78)$, etc.

Option 4. In some cases, Pippenger's algorithm **clumps the output**. Figure 9 summarizes output clumping.

Output clumping is the transpose of input clumping. It works as follows. Choose an integer $b \geq 2$ and an integer $a \geq b$. Partition the *output* indices $\{1, 2, 3, \ldots, q\}$ into $\lceil q/a \rceil$ parts, each part of size a or smaller: $Q_0 = \{1, 2, \ldots, a\}$ and $Q_1 = \{a+1, a+2, \ldots, 2a\}$ and so on.

$$
\begin{array}{c}
p \text{ inputs: } x_1, x_2, \ldots, x_p \\
\Big\downarrow \text{recursively; weight at most } \lfloor w/b \rfloor \\
\text{at most } \lceil q/a \rceil \binom{a}{b} \text{ relevant } X(T(U))\text{'s} \\
\Big\downarrow \text{at most } \lceil q/a \rceil \binom{a}{b} (b-1) \text{ multiplications} \\
\text{at most } q \text{ relevant products of } X(T(U))\text{'s} \\
\Big\downarrow \text{at most } p \lceil q/a \rceil (b-1) \text{ multiplications} \\
q \text{ outputs}
\end{array}
$$

FIGURE 9. Output clumping.

Consider all the size-b subsets U of parts. For each $i \in \{1, 2, \ldots, p\}$, choose a decomposition of $\{j : i \in S_j\}$ as a disjoint union of as many U's as possible, plus a few overflow elements—at most $b - 1$ overflow elements in each part, totaling at most $\lceil q/a \rceil (b - 1)$ overflow elements. Define $T(U)$ as the set of i's for which U is used in this union.

Use Pippenger's algorithm recursively to compute $X(T(U))$ for each U such that $T(U)$ is nonempty; the weight here is at most $\lfloor w/b \rfloor$. Then compute each desired $X(S_j)$ as the product of $X(T(U))$ for each U containing j and any relevant overflow x's; there are at most $\lceil q/a \rceil \binom{a}{b} (b - 1)$ multiplications of $X(T(U))$'s, and then at most $p \lceil q/a \rceil (b - 1)$ overflow multiplications.

Example: Consider one last time $134567, 1235, 145678, 234578, 124568, 123478, 1234568, 1357, 2348, 13567$. Choose $b = 2$ and $a = 4$. Use Pippenger's algorithm recursively to compute $X(135)$, $X(467)$, $X(2)$, $X(58)$, etc. The desired outputs are $X(135)X(467)$, $X(135)X(2)$, $X(467)X(58)x_1$, $X(2)X(58)x_3x_4x_7$, etc.

The complete algorithm. Pippenger's algorithm is parametrized by a recursion level L, a sequence $c, a_1, b_1, a_2, b_2, \ldots, a_{L-1}, b_{L-1}$, and an optional toggle. It first partitions the input with parameter c, then clumps the output with parameters a_1, b_1 at the next level of recursion, then clumps the input with parameters a_2, b_2 at the next level of recursion, and so on, alternating between clumping the output and clumping the input. Finally, after L levels of recursion, Pippenger's algorithm simply computes each desired product separately.

The effect of the toggle is to transpose everything: partition the output, then clump the input, then clump the output, etc.

Later I will discuss recursion level 1, with parameter c, and recursion level 2, with parameters c, a, b. I have left the toggle—in particular, output partitioning—as an exercise for the reader.

Pippenger did much more optimization work in [34] and [35]. Pippenger proved that a particular parameter sequence uses at most $(1+o(1))pq/\lg pq$ multiplications, if $(\lg p)/q$ and $(\lg q)/p$ are both in $o(1)$. Pippenger also showed, by adapting a counting argument of Lupanov in [29], that almost all sequences of q subsets require $(1 + o(1))pq/\lg pq$ multiplications, if $(\lg p)/q$ and $(\lg q)/p$ are both in $o(1)$.

One can quickly compute Pippenger's parameter sequence given p, q, although Pippenger did not say this explicitly. See [34] or [35] for details of the construction. In practice, one can usually afford to experiment with many parameter sequences.

Recursion level 1. The special case $L = 1$ of Pippenger's algorithm partitions the input with parameter c, and computes each desired product separately inside the recursion. This takes at most $\lceil p/c \rceil (2^c - c - 1) + \lceil p/c \rceil q - q$ multiplications; see Figure 7.

The choice $c \approx \lg q - \lg\lg q$ is reasonable if q is fairly large and p is substantially larger than $\lg q$. Then $2^c - c - 1 \approx q/\lg q$ and $\lceil p/c \rceil \approx p/\lg q$, so there are at most about $pq/\lg q + pq/(\lg q)^2$ multiplications.

Recursion level 2. The special case $L = 2$ of Pippenger's algorithm partitions the input with parameter c, clumps the output with parameters a and b inside the recursion, and computes each desired product separately at the next level of recursion.

The choices $c \approx \lg q - 3\lg\lg q$ and $a \approx q/(\lg q)^2$ and $b = 2$ are reasonable if q is fairly large and $p \geq q$. Define $p' = pq/(\lg q)^4$ and $w' = pq/\lg q$. The input partitioning takes at most about p' multiplications to reduce to a problem of size at most about p', q, w'; see Figure 7. The output clumping takes at most about $q^2/2(\lg q)^2 + p'(\lg q)^2$ multiplications to reduce to a problem of weight at most about $w'/2$; see Figure 9. The rest of the computation takes at most about $w'/2$ multiplications. The total number of multiplications is at most about $pq/2\lg q + pq/(\lg q)^2 + q^2/2(\lg q)^2 + pq/(\lg q)^4$.

7. Pippenger's exponentiation algorithm, 1976

Pippenger's exponentiation algorithm in [34] reduces any exponentiation problem to a multiple-product problem, and then solves that problem with the algorithm described in Section 6.

Single base. Given x, a bound B, and a set S of integers in $\{1, 2, \ldots, B-1\}$, Pippenger's exponentiation algorithm computes x^e for all $e \in S$ as follows.

The first step is the same as in Yao's algorithm. Choose a positive integer k. Define $x_i = x^{2^{ki}}$ and $p = \lceil (\lg B)/k \rceil$. Compute $x_0, x_1, x_2, \ldots, x_{p-1}$ with $(p-1)k$ squarings.

Express each desired exponent $e \in S$ as $e_0 + 2e_1 + \cdots + 2^{k-1}e_{k-1}$, where each e_j is a sum of distinct powers of 2^k. In other words, write e in radix 2^k, write each coefficient in radix 2, and transpose the resulting matrix of bits.

By construction, each x^{e_j} for $e_j \neq 0$ is a product of a nonempty subsequence of $x_0, x_1, \ldots, x_{p-1}$. There are at most $k\#S$ of these products; compute all of them by Pippenger's multiple-product algorithm.

Finally, compute each x^e as $((\cdots(x^{e_{k-1}})^2 x^{e_{k-2}} \cdots)^2 x^{e_1})^2 x^{e_0}$. There are at most $2(k-1)$ multiplications here, totaling $2(k-1)\#S$ for all $e \in S$.

Consider, for example, exponents 31415 and 27182. Choose $k = 3$. Express 31415 as $e_0 + 2e_1 + 4e_2$ where $e_0 = 2^0 + 2^9 + 2^{12}$, $e_1 = 2^0 + 2^3 + 2^6 + 2^{12}$, and $e_2 = 2^0 + 2^3 + 2^9 + 2^{12}$; express 27182 as $f_0 + 2f_1 + 4f_2$ where $f_0 = 2^3 + 2^9$, $f_1 = 2^0 + 2^{12}$, and $f_2 = 2^0 + 2^3 + 2^9 + 2^{12}$. Compute $x^{2^0}, x^{2^3}, x^{2^6}, x^{2^9}, x^{2^{12}}$; then compute $x^{e_0}, x^{e_1}, x^{e_2}, x^{f_0}, x^{f_1}, x^{f_2}$; finally compute x^{31415} and x^{27182}. Figure 10 shows an addition chain obtained in this way.

1	=	1_2	=	2^0
2	=	10_2		
4	=	100_2		
8	=	1000_2	=	2^3
16	=	10000_2		
32	=	100000_2		
64	=	1000000_2	=	2^6
128	=	10000000_2		
256	=	100000000_2		
512	=	1000000000_2	=	2^9
1024	=	10000000000_2		
2048	=	100000000000_2		
4096	=	1000000000000_2	=	2^{12}
520	=	1000001000_2	=	$2^3+2^9=f_0$
4097	=	1000000000001_2	=	$2^0+2^{12}=f_1$
4161	=	1000001000001_2	=	$2^0+2^6+2^{12}$
4169	=	1000001001001_2	=	$2^0+2^3+2^6+2^{12}=e_1$
4609	=	1001000000001_2	=	$2^0+2^9+2^{12}=e_0$
4617	=	1001000001001_2	=	$2^0+2^3+2^9+2^{12}=e_2=f_2$
9234	=	10010000010010_2	=	$2e_2$
13403	=	11010001011011_2	=	$2e_2+e_1$
26806	=	110100010110110_2	=	$4e_2+2e_1$
31415	=	111101010110111_2	=	$4e_2+2e_1+e_0$
9234	=	10010000010010_2	=	$2f_2$
13331	=	11010000010011_2	=	$2f_2+f_1$
26662	=	110100000100110_2	=	$4f_2+2f_1$
27182	=	110101000101110_2	=	$4f_2+2f_1+f_0$

FIGURE 10. Computing 27182 and 31415 with 26 additions.

Pippenger proved that, with a particular choice of parameters, this algorithm computes x^e for all $e \in S$ with at most $\lg B + (1+o(1))(\#S \lg B)/\lg(\#S \lg B)$ multiplications, if $\lg \#S$ is in $o(\lg B)$. Pippenger chose $k = \lceil\sqrt{(\lg B)/\#S}\rceil$: for example, $k=1$ if $\#S \ge \lg B$. There are $(p-1)k < \lg B$ initial squarings; at most $2(k-1)\#S < 2\sqrt{\#S \lg B}$ final multiplications; and, in the middle, at most about $pk\#S/\lg(pk\#S) \approx \#S \lg B/\lg(\#S \lg B)$ multiplications to compute at most $k\#S$ products of subsequences of $x_0,\dots,x_{p-1}$.

Single base, recursion level 1. Here is a complete description of one special case of Pippenger's exponentiation algorithm, with recursion level $L=1$ in Pippenger's multiple-product algorithm.

The inputs are x, a bound B, and a set S of integers in $\{1,2,\dots,B-1\}$. The algorithm will compute x^e for all $e \in S$.

Choose a positive integer k. Define $x_i = x^{2^{ki}}$ and $p = \lceil(\lg B)/k\rceil$. Compute $x_0, x_1, \dots, x_{p-1}$ with $(p-1)k$ squarings.

Choose a positive integer c. For each j, compute $\prod_{i \in T} x_i$ for all nonempty subsets T of $\{cj, cj+1, \dots, cj+c-1\} \cap \{0,1,\dots,p-1\}$. There are at most $2^c - 1$

such sets, totaling at most $\lceil p/c \rceil (2^c - 1)$ over all j; this computation takes at most $\lceil p/c \rceil (2^c - c - 1)$ multiplications.

Now compute each desired x^e with at most $k(\lceil p/c \rceil - 1) + 2(k-1)$ multiplications as follows. Write e as $e_0 + 2e_1 + \cdots + 2^{k-1}e_{k-1}$, where each e_j is a sum of distinct powers of 2^k. Compute x^{e_j} as a product of at most $\lceil p/c \rceil$ of the previously computed products $\prod_{i \in T} x_i$. Then compute x^e as $((\cdots(x^{e_{k-1}})^2 x^{e_{k-2}} \cdots)^2 x^{e_1})^2 x^{e_0}$.

Single base, recursion level 1, with $k = 1$. For example, if $k = 1$, simply write each new exponent e in radix 2^c as $f_0 + 2^c f_1 + \cdots$, and compute x^e as the product of x^{f_0} and $x^{2^c f_1}$ and so on, all of which have been previously computed.

The initial computation of $\lceil p/c \rceil (2^c - 1) \approx 2^c (\lg B)/c$ products takes $p - 1 + \lceil p/c \rceil (2^c - c - 1) \approx 2^c (\lg B)/c$ multiplications. Each new exponent then takes just $\lceil p/c \rceil - 1 \approx (\lg B)/c$ multiplications.

Apparently this special case was, historically, the first part of Pippenger's work. The total number of multiplications, $(\#S + 2^c)(\lg B)/c$, was reported by Yao in [40] as an improvement by Pippenger upon Yao's results. If $c \approx \lg \#S - \lg\lg \#S$ and $\#S$ is large then this special case of Pippenger's algorithm uses about $\#S(\lg B)/\lg \#S$ multiplications, while Yao's algorithm uses about $\#S(\lg B)/\lg\lg B$ multiplications.

The general case. Pippenger's exponentiation algorithm, in its full generality, computes q products of powers of $x_1, x_2, \ldots, x_p$ as follows. Write each desired output in the form $y_0 y_1^2 \cdots y_{k-1}^{2^{k-1}}$, where each y_h is a product of $x_j^{2^{ik}}$ for some set of pairs (i, j). Compute all $x_j^{2^{ik}}$ by successive squarings; compute all y_h by Pippenger's multiple-product algorithm; then compute each desired output with at most $2(k-1)$ multiplications.

Assume that each exponent is below B, and that $(\lg P)/q \lg B$ and $(\lg q)/p \lg B$ are in $o(1)$. Pippenger proved in [34] that, with a particular choice of parameters, this algorithm uses at most $p \lg B + (1 + o(1))(pq \lg B)/\lg(pq \lg B)$ multiplications, and a transposed algorithm uses at most $q \lg B + (1 + o(1))(pq \lg B)/\lg(pq \lg B)$ multiplications.

By generalizing the counting arguments of Lupanov in [29] and Erdős in [17], Pippenger showed in [36] that almost all exponent matrices require $\min\{p, q\} \lg B + (1 + o(1))(pq \lg B)/\lg(pq \lg B)$ multiplications. Thus there is not much room for improvement in Pippenger's algorithm. Of course, minor improvements may still be useful in practice.

Setting the record straight. The special case $L = 1$ of Pippenger's algorithm is often incorrectly credited to Lim and Lee, who reinvented it many years later and published it in [28], after Brickell, Gordon, McCurley, and Wilson in [10] reinvented and published the special case $L = k = 1$.

Knuth in [22, answer to exercise 4.6.3–39] described Pippenger's results as a "comprehensive generalization" of Straus's results and Yao's results. This is rather misleading. I did not realize for many years that—for example—Pippenger had a different, and generally faster, algorithm to compute many powers of a single base.

References

[1] —, *17th annual symposium on foundations of computer science*, IEEE Computer Society, Long Beach, California, 1976. MR 56 #1766.

[2] Andreas Antoniou, *Digital filters: analysis and design*, McGraw-Hill, New York, 1979. ISBN 0070021171.

[3] 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. MR 42 #4441.

[4] Edward G. Belaga, *The additive complexity of a natural number*, Soviet Mathematics Doklady **17** (1976), 5–9. MR 53 #13141.

[5] G. R. Blakley, David Chaum (editors), *Advances in cryptology: CRYPTO '84*, Lecture Notes in Computer Science, 196, Springer-Verlag, Berlin, 1985. ISBN 3–540–15658–5. MR 86j:94003.

[6] Irina E. Bocharova, Boris D. Kudryashov, *Fast exponentiation in cryptography*, in [12] (1995), 146–157. MR 97m:94013.

[7] Jurjen Bos, Matthijs Coster, *Addition chain heuristics*, in [8] (1989), 400–407.

[8] Gilles Brassard (editor), *Advances in cryptology—CRYPTO '89*, Lecture Notes in Computer Science 435, Springer-Verlag, Berlin, 1990. ISBN 0–387–97317–6. MR 91b:94002.

[9] Alfred Brauer, *On addition chains*, Bulletin of the American Mathematical Society **45** (1939), 736–739. MR 1,40a.

[10] Ernest F. Brickell, Daniel M. Gordon, Kevin S. McCurley, David B. Wilson, *Fast exponentiation with precomputation (extended abstract)*, in [37] (1993), 200–207; newer version in [11].

[11] Ernest F. Brickell, Daniel M. Gordon, Kevin S. McCurley, David B. Wilson, *Fast exponentiation with precomputation: algorithms and lower bounds* (1995); draft in [10]. Available from `http://research.microsoft.com/~dbwilson/bgmw/`.

[12] Girard Cohen, Marc Giusti, Teo Mora (editors), *Applied algebra, algebraic algorithms and error-correcting codes*, Lecture Notes in Computer Science 948, Springer-Verlag, Berlin, 1995. ISBN 3–540–60114–7. MR 97k:68003.

[13] Yvo Desmedt (editor), *Advances in cryptology—CRYPTO '94*, Lecture Notes in Computer Science 839, Springer-Verlag, Berlin, 1994.

[14] Peter Downey, Benton Leong, Ravi Sethi, *Computing sequences with addition chains*, SIAM Journal on Computing **10** (1981), 638–646. MR 82h:68064.

[15] Taher ElGamal, *A public key cryptosystem and a signature scheme based on discrete logarithms*, in [5] (1985), 10–18; newer version in [16]. MR 87b:94037.

[16] Taher ElGamal, *A public key cryptosystem and a signature scheme based on discrete logarithms* **31** (1985), 469–472; draft in [15]. MR 86j:94045.

[17] Paul Erdős, *Remarks on number theory III: On addition chains*, Acta Arithmetica **6** (1960), 77–81. MR 22 #12085.

[18] Charles M. Fiduccia, *On obtaining upper bounds on the complexity of matrix multiplication*, in [32] (1972), 31–40. MR 52 #12398.

[19] Ronald L. Graham, Andrew C. Yao, Frances F. Yao, *Addition chains with multiplicative cost*, Discrete Mathematics **23** (1978), 115–119. MR 80d:68051.

[20] Michael Kaminski, David G. Kirkpatrick, Nader H. Bshouty, *Addition requirements for matrix and transposed matrix products*, Journal of Algorithms **9** (1988), 354–364. MR 89m:68061.

[21] Donald E. Knuth, *The art of computer programming, volume 2: seminumerical algorithms*, 1st edition, 2nd printing, Addison-Wesley, Reading, 1971. MR 44 #3531.

[22] Donald E. Knuth, *The art of computer programming, volume 2: seminumerical algorithms*, 2nd edition, Addison-Wesley, Reading, 1981. ISBN 0–201–03822–6. MR 83i:68003.

[23] Donald E. Knuth, *The art of computer programming, volume 2: seminumerical algorithms*, 3rd edition, Addison-Wesley, Reading, 1997. ISBN 0–201–89684–2.

[24] Donald E. Knuth (editor), *Selected papers on analysis of algorithms*, CSLI Publications, Stanford, 2000. ISBN 1–57586–212–3. MR 2001c:68066.

[25] Donald E. Knuth, Christos H. Papadimitriou, *Duality in addition chains*, Bulletin of the European Association for Theoretical Computer Science **13** (1981), 2–4; reprinted in [24, chapter 31].

[26] K. Y. Lam, L. C. K. Hui, *On the efficiency of $SS(l)$ square-and-multiply exponentiation algorithms*, Electronics Letters **30** (1994), 2115–2116.

[27] Adrien-Marie Legendre, *Recherches d'analyse indéterminée*, Histoire de L'Académie Royale des Sciences (1785), 465–559.

[28] Chae Hoon Lim, Pil Joong Lee, *More flexible exponentiation with precomputation*, in [13] (1994), 95–107.

[29] O. B. Lupanov, *On rectifier and contact rectifier circuits*, Doklady Akademii Nauk SSSR **111** (1956), 1171–1174. MR 19,807a.

[30] Daniel P. McCarthy, *The optimal algorithm to evaluate x^n using elementary multiplication methods*, Mathematics of Computation **31** (1977), 251–256. MR 55 #1811.

[31] Daniel P. McCarthy, *Effect of improved multiplication efficiency on exponentiation algorithms derived from addition chains*, Mathematics of Computation **46** (1986), 603–608. MR 87e:68046.

[32] Raymond E. Miller, James W. Thatcher (editors), *Complexity of computer computations*, Plenum Press, New York, 1972. ISBN 0306307073. MR 51 #9575.

[33] Jorge Olivos, *On vectorial addition chains*, Journal of Algorithms **2** (1981), 13–21. MR 83h:68044.

[34] Nicholas Pippenger, *On the evaluation of powers and related problems (preliminary version)*, in [1] (1976), 258–263; newer version split into [35] and [36]. MR 58 #3682.

[35] Nicholas Pippenger, *The minimum number of edges in graphs with prescribed paths*, Mathematical Systems Theory **12** (1979), 325–346; draft in [34]. MR 81e:05079.

[36] Nicholas Pippenger, *On the evaluation of powers and monomials*, SIAM Journal on Computing **9** (1980), 230–250; draft in [34]. MR 82c:10064.

[37] Rainer A. Rueppel (editor), *Advances in cryptology: EUROCRYPT '92*, Lecture Notes in Computer Science 658, Springer-Verlag, Berlin, 1993. ISBN 3–540–56413–6. MR 94e:94002.

[38] Ernst G. Straus, *Addition chains of vectors (problem 5125)*, American Mathematical Monthly **70** (1964), 806–808.

[39] Edward G. Thurber, *On addition chains $l(mn) \leq l(n) - b$ and lower bounds for $c(r)$*, Duke Mathematical Journal **40** (1973), 907–913. MR 48 #8429.

[40] Andrew C. Yao, *On the evaluation of powers*, SIAM Journal on Computing **5** (1976), 100–103. MR 52 #16128.

[41] Hans Zantema, *Minimizing sums of addition chains*, Journal of Algorithms **12** (1993), 281–307. MR 92i:68064.

Department of Mathematics, Statistics, and Computer Science (M/C 249), The University of Illinois at Chicago, Chicago, IL 60607–7045

E-mail address: `djb@cr.yp.to`