Berry Schoenmakers
Department of Mathematics & Computer Science
Eindhoven University of Technology
berry@win.tue.nl

Binary pebbling algorithms for in-place reversal of one-way hash chains

Berry Schoenmakers is associate professor (UHD) in the Coding and Crypto Group at Eindhoven University of Technology. Schoenmakers is known for his work on cryptographic protocols for electronic voting, electronic payment, and secure computation. In this article he discusses an algorithmic problem pertaining to the backward traversal of a one-way hash chain, which has a unique motivation in cryptography. At the core of its recently obtained solution lies an intricate fractal structure which turns out to have a very nice and simple characterization.

The problem is formulated in terms of a length-preserving one-way function f. A concrete example is the classical Davies–Meyer one-way function constructed from a block cipher such as AES:

$$f : \begin{cases} \{0,1\}^{128} \to \{0,1\}^{128} \\ \qquad x \mapsto \mathrm{AES}_x(\mathbf{0}). \end{cases}$$

That is, $f(x)$ is computed as an AES encryption of the trivial all-zero message $\mathbf{0}$ under the key x, which can obviously be done efficiently. On the other hand, recovering x from $f(x)$ is tantamount to recovering an AES key given a single plaintext–ciphertext pair, which is assumed to be computationally hard. Therefore, f is called a *one-way function*, as it is easy to evaluate but hard to invert. Block ciphers like AES are normally used for symmetric encryption to provide confidentiality, whereas one-way functions like f are often used for asymmetric authentication, e.g., in the construction of digital signature schemes.

One-way chains

Back in 1981, Lamport (the 'La" in LaTeX) proposed an elegant asymmetric identification scheme which operates in terms of one-way chains [12]. A *one-way chain* is the sequence formed by the successive iterates of f for a given value. For example, in a client-server setting, the client may apply f four times for a randomly chosen 128-bit seed value x_0 to obtain a length-4 chain:

$$x_0 \xrightarrow{f} x_1 \xrightarrow{f} x_2 \xrightarrow{f} x_3 \xrightarrow{f} x_4.$$

Lamport's *identification scheme* then operates as follows. At the start, the client registers itself securely with the server, as a result of which the server associates the endpoint x_4 with the client. Depending on the details, this registration step may be rather involved. However, from now on the client may identify itself securely to the server simply by releasing the next preimage on the chain. In the first round of identification, the client releases preimage x_3, and the server checks this value by testing if $f(x_3) = x_4$ holds. An eavesdropper obtaining x_3 cannot impersonate the client later on because the next round the server will demand a preimage for x_3. At this stage, preimage x_2 satisfying $f(x_2) = x_3$ is known only to the client; it is not even known to the server yet, which is why the scheme provides *asymmetric* authentication.

One-way chains and variations thereof are often referred to as *hash chains* since cryptographic hash functions such as SHA-256 are commonly used as alternatives for f. Hash chains are fundamental to many constructions in cryptography, and even to some forms of cryptanalysis (e.g., rainbow tables). Bitcoin's blockchain [15] is probably the best-known example of a hash chain nowadays — but note that blockchains are costly to generate due to the additional 'proof of work' requirement for the hash values linking successive blocks. Hash chains are also used in digital signature schemes required to be quantum secure, building on work by Merkle from 1979 [14]. Incidentally, Merkle attributes the use of iterated functions to Winternitz. However, Winternitz's idea is to use only one preimage on a length-n chain, basically to securely encode an integer in the

set $\{0,\ldots,n-1\}$, whereas Lamport's idea is to use all of the n preimages. The CAFE phone-tick scheme [2, Section 3.5] (see also [17]) and later micropayment schemes (e.g., PayWord [19]) actually combine these two ideas. In the case of phone-ticks, the caller releases the endpoint of a chain at the start of a call; at each tick, the caller simply releases the next preimage (as in Lamport's scheme) to pay for continuing the call. After the call ends, the phone company only needs to keep the last preimage released by the caller to claim the amount due (as in Winternitz's encoding).

Security of one-way chains

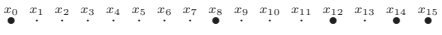
The use of a cryptographic hash function to create a one-way chain is overkill, however. A function like SHA-256 is not just one-way but is also designed to compress bit strings of practically unlimited length, and related to this, SHA-256 is required to be collision-resistant as well. For the security of a one-way chain, f should be one-way, that is, given y in the range of f it must be hard to find any x such that $f(x)=y$. Or rather, as recognized in [13,17], f should necessarily be *one-way on its iterates*, which says that, for a length-n chain, given an nth iterate image y (in the range of f^n) it must be hard to find any x such that $f(x)=y$.

Viewing f as a random function (as in the random oracle model for hash functions), it follows that finding such a preimage x takes $2^{128}/n$ time approximately. If $n=1$ this is simply the problem of inverting f, which can only be solved by making random guesses for x; on each attempt one succeeds with probability $1/2^{128}$. For $n>1$, however, one should not guess randomly. First, observe that the set of nth iterate images y (range of f^n) is much smaller than $\{0,1\}^{128}$. In fact, the expected number of nth iterate images y is equal to $(1-\tau_n)2^{128}$, where $\tau_0=0$, $\tau_n=e^{-1+\tau_{n-1}}$ for $n\geq 1$ [4, Theorem 2(v)]. To take advantage of the given that y is not just any image but an nth iterate image, start with a random guess x_0 and then check if $x_1=f(x_0)$ happens to match y. Next, compute $x_2=f(x_1)$ and again test for equality with y, and continue to do so until x_n is reached. The overall probability of hitting y and thus obtaining a preimage of y as well works out as $n/2^{128}$ approximately. Hence, even for very long chains of length $n=2^{32}$, say, the security level is still 2^{96}. See also [8, Theorem 3] for a further analysis.

Pebbling algorithms

The above provides a solid basis for Jakobsson's wonderful idea of using efficient pebbling algorithms to make Lamport's scheme practical even for very long chains [11]. Naive implementations would render Lamport's scheme completely impractical: both (i) computing $x_{n-1}=f^{n-1}(x_0)$ to perform the first round of identification, then computing $x_{n-2}=f^{n-2}(x_0)$ from scratch, and so on, and (ii) storing all of $x_0,x_1,\ldots,x_{n-2},x_{n-1}$ to perform each round of identification instantly, are out of the question. The crux of Jakobsson's pebbling algorithm is to achieve a good *space-time trade-off*: for chains of length $n=2^k$, Jakobsson's algorithm stores $O(\log n)$ hash values throughout, and the maximum number of hashes performed in any round of identification is $O(\log n)$ as well.

Each hash value stored is associated with a *pebble*. For a length-16 chain, five pebbles are initially arranged as follows, which is typical of a *binary pebbling algorithm*:



The general pattern is that starting from the rightmost pebble, the distance to the next pebble doubles each time. From this initial arrangement, the first two elements x_{15} and x_{14} of the reverse of $\{x_0,x_1,\ldots,x_{15}\}$ can be output directly. For the third element x_{13} we need to apply f once to recompute it from x_{12}. The fourth element x_{12} can be output again without any effort.

To produce x_{11}, something interesting happens. Because f is one-way, the only sensible option is to recompute it from x_8 as $x_{11}=f^3(x_8)$. But while doing so, the value of $x_{10}=f^2(x_8)$ is also stored for later use. Hence, just before x_{11} is output, the pebbles are arranged as follows:

Proceeding this way and computing outputs just-in-time, the *rushing* binary pebbling algorithm R_k is obtained:

$$R_0(x) = \text{output } x,$$
$$R_k(x) = R_{k-1}(f^{2^{k-1}}(x)); R_{k-1}(x).$$

The reader may check that $R_k(x)$ outputs the sequence

$$f_k^*(x) = \{f^i(x)\}_{i=0}^{2^k-1}$$

in reverse, using $k2^{k-1}$ hashes in total. In addition, the storage requirements are low: $R_k(x)$ needs to store x for the recursive call $R_{k-1}(x)$ later on, which leads to a maximum of $k+1$ values stored (pebbles) at any moment.

The only drawback is that R_k in the worst case requires time exponential in k between producing successive outputs. Removing these slow rounds is exactly what makes the problem non-trivial. That is, we seek a way to reverse $f_k^*(x)$ satisfying the performance constraints of using $O(k)$ storage (pebbles) and using $O(k)$ applications of f (hashes) between producing *any* two successive outputs. To study this problem we introduce a specific framework for binary pebbling algorithms that operate in rounds.

At this point we like to mention that there are many similar notions of 'pebbling' in the literature. In particular, pebbling games (see, e.g., [16]) are somewhat related, and have recently been used in the context of cryptography to prove memory-hardness of certain hash functions [1]. Graph pebbling is another well-known problem (see, e.g., [9]). Reversible computing (see, e.g., [18]) gives rise to even more uses of pebbling (aka 'checkpointing', see below). As discussed in [20], however, the specific worst-case constraint limiting the number of hashes per round is unique to the cryptographic setting, starting with the work in [10,11].

Framework for binary pebbling

For $k\geq 0$, $P_k(x)$ will be defined as an algorithm that runs for $2^{k+1}-1$ rounds in total, and outputs $f_k^*(x)$ in reverse in its last 2^k rounds. It is essential that we include the initial 2^k-1 rounds (in which no outputs are produced) as an integral part of pebbler $P_k(x)$, as this allows for a fully recursive definition and analysis of binary pebbling. In fact, in terms of a given *schedule* $T_k=\{t_{k,r}\}_{r=1}^{2^k-1}$, which fixes the number of hashes for each initial round, a *binary pebbler* $P_k(x)$ is completely specified by the recursive definition given in Figure 1. This means, for example, that $P_0(x)$ runs for one round only outputting $y_0=x$ itself, and that $P_1(x)$ will run for three rounds, performing $t_{1,1}=1$ hash in its first round, outputting $y_0=f(x)$ in its second round, and outputting $y_1=x$ in its last round. In general, $P_k(x)$ computes $f^{2^k-1}(x)$ using exactly 2^k-1 hashes in total in its initial stage, storing only the values $y_k,\ldots,y_0$ along the way. Running pebblers

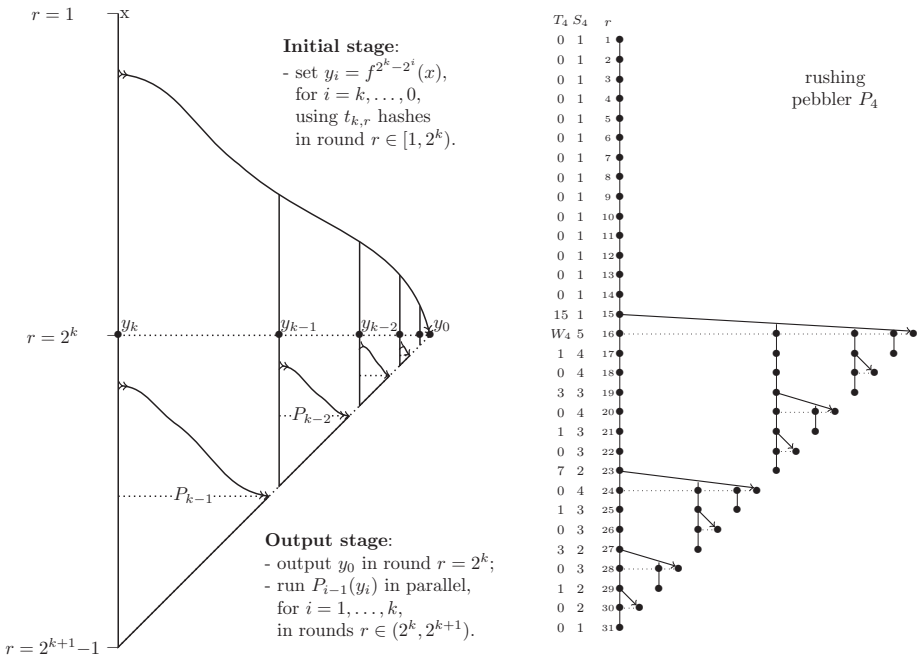


Figure 1 Binary pebbler $P_k(x)$ for schedule $T_k = \{t_{k,r}\}_{r=1}^{2^k-1}$ satisfying $\sum_{r=1}^{2^k-1} t_{k,r} = 2^k - 1$ (left). Schedule T_4, work W_4, and storage S_4 for rushing pebbler P_4 in rounds $r = 1$ to $r = 31$ (right). Bullets represent stored values (pebbles), rightwards arrows represent hashing, vertical lines represent copying.

$P_{k-1}, \ldots, P_0$ in parallel in the output stage means that pebblers take turns to execute for one round each, where the order in which this happens within a round is irrelevant. It is not hard to prove that in every round exactly one of the pebblers running in parallel will be in its first output round, and that the sequence of outputs is always equal to $f_k^*(x)$.

Schedule T_k specifies the number of hashes for the initial stage of P_k. To analyze the *work* done by P_k in its output stage, we let sequence W_k of length $2^k - 1$ denote the number of hashes performed by P_k in each of its last $2^k - 1$ rounds — noting that by definition no hashes are performed by P_k in round 2^k. The following recurrence relation for W_k will be useful throughout:

$$W_0 = \{\}, \quad W_k = T_{k-1} + W_{k-1} \| \{0\} \| W_{k-1},$$

where $T_{k-1} + W_{k-1}$ denotes elementwise addition of T_{k-1} and W_{k-1} and $\|$ concatenation of sequences (+ takes precedence over $\|$).

To analyze the *storage* needed by P_k the number of hash values stored by P_k will be counted for each round. We let sequence $S_k = \{s_{k,r}\}_{r=1}^{2^{k+1}-1}$ denote the total storage used by P_k at the start of each round. For instance, $s_{k,1} = 1$ as P_k only stores x at the start, and $s_{k,2^k} = k + 1$ as P_k stores $y_0, \ldots, y_k$ at the start of round 2^k independent of schedule T_k.

The rushing pebbler P_k corresponding to R_k introduced above is obtained by taking schedule T_k with $t_{k,2^k-1} = 2^k - 1$ and $t_{k,r} = 0$ elsewhere. Rushing pebbler P_4 is illustrated in Figure 1 in our framework for binary pebbling. The storage S_4 is minimal throughout, but for the work W_4 there are big peaks: e.g., in round 23, in total seven hashes are performed, while the pebbler is idle in all even rounds.

Towards optimal solution

As it turns out, our framework admits a simple solution obtained by taking schedule $T_k = \{1\}_{r=1}^{2^k-1}$, resulting in the *speed-1* pebbler illustrated in Figure 2. The above recurrence relation for W_k yields $\max(W_k) = k - 1$ for $k \geq 1$, and it can also be shown that $\max(S_k) = \max(k + 1, 2k - 2) = O(k)$. The speed-1 pebbler thus achieves the desired asymptotic bounds. For practical purposes, however, further savings are needed to limit the costs as much as possible. E.g., to enable a lightweight client device to identify itself every half hour for a period of three years using a length-2^{16} chain.

Jakobsson's pebbling algorithm [11] provides a clever way to cut storage $\max(S_k)$ in half essentially. Translated to our framework for binary pebbling, the corresponding schedule T_k is obtained by setting $t_{k,r} = 0$ for $1 \leq r < 2^{k-1}$, $t_{k,r} = 2$ for $2^{k-1} \leq r < 2^k - 1$, and $t_{k,2^k-1} = 1$. The pebbler obtained this way is called the *speed-2* pebbler, illustrated in Figure 2.

Compared to a speed-1 pebbler, the crucial idea of a speed-2 pebbler is to remain idle for the first half of the initial stage — preventing that too many pebblers are active at the same time — and then make up for this by hashing at double speed in the remaining time. It can be proved that $\max(W_k) = k - 1$ for $k \geq 1$ also holds for a speed-2 pebbler, but compared to a speed-1 pebbler storage is now reduced by 50%, achieving $\max(S_k) = k + 1$. For binary pebbling algorithms, storage S_k of up to $k + 1$ hash values is optimal, since this amount of storage is already needed during the first output round $r = 2^k$, for any binary pebbler P_k. The interesting question is whether the work $\max(W_k)$ can be reduced any further?

Optimal binary pebbling

An elementary analysis yields $\max(W_k) \geq \lceil k/2 \rceil$, $k \geq 2$, as lower bound for any binary pebbling algorithm (see [20, Theorem 2]). So, the best that can be achieved is to reduce the maximum number of hashes for any output round to $k/2$ roughly. The problem of optimally efficient hash chain reversal was extensively studied by Coppersmith and Jakobsson [3]. They achieved nearly optimal space-time complexity for a complicated pebbling algorithm using $k + \lceil \log_2(k+1) \rceil$ pebbles and no more than $\lfloor \frac{k}{2} \rfloor$ hashes per round. Hence, an excess storage of approximately $\log_2 k$ hash values compared to optimal binary pebbling.

Fortunately, Yum et al. [21] observed that a greedy implementation of Jakobsson's original pebbling algorithm already achieves the optimal space-time trade-off for binary pebbling. Their idea is to greedily use up a budget of $\lfloor \frac{k}{2} \rfloor$ hashes per round subject to the constraint that no more than about k hash values are stored at any time. The only drawback of the greedy approach is that no apparent structure is revealed.

In contrast, we have found an explicit, essentially unique solution for optimal binary pebbling, which leads to a complete understanding of the problem and paves the way for fully optimized in-place implementations. As a closed formula, the optimal schedule T_k is obtained by setting $t_{k,r} = 0$ for $1 \leq r < 2^{k-1}$, and setting $t_{k,r}$ to

$$\left\lfloor \tfrac{1}{2}((k+r) \bmod 2 + k + 1 - \mathrm{len}((2r) \bmod 2^{\mathrm{len}(2^k - r)})) \right\rfloor$$

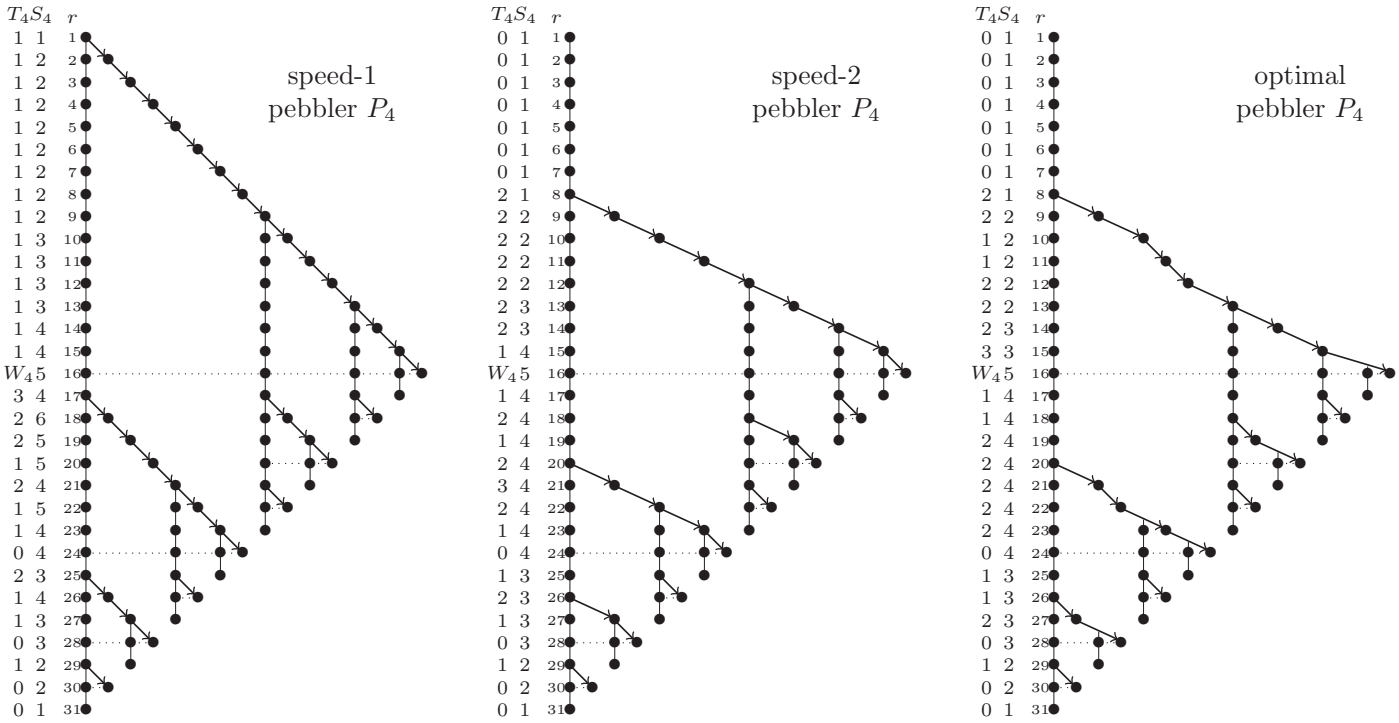


Figure 2 Schedule T_4, work W_4, storage S_4 for three types of binary pebblers P_4 in rounds 1–31

for $2^{k-1} \le r < 2^k$, where $\mathrm{len}(n)$ denotes the bit length of nonnegative integer n. Optimal pebbler P_4 is illustrated in Figure 2, which uses the following optimal schedules:

$$T_0 = \{\},$$
$$T_1 = \{1\},$$
$$T_2 = \{0,1,2\},$$
$$T_3 = \{0,0,0,2,1,2,2\},$$
$$T_4 = \{0,0,0,0,0,0,0,2,2,1,1,2,2,2,3\}.$$

In general, an optimal pebbler P_k will use up to $\max(W_k) = \lceil k/2 \rceil$ hashes in any output round. For the optimal pebbler P_4 in Figure 2, this works out as $\max(W_4) = 2$ hashes, compared to the speed-2 pebbler P_4 which needs 3 hashes in output round $r = 21$.

The fractal nature of the optimal schedule T_k is revealed by the recursive characterization in terms of sequences U_k, V_k defined in Table 1. These sequences are defined over $\frac{1}{2}\mathbb{Z}$ — rather than over $\mathbb{Z}$ as will ultimately be required for use in a pebbling algorithm. Without rounding of these half-integers, the optimal schedule satisfies the following *key equation* in terms of sequences U_k, V_k, $k \ge 2$:

$$(U_k \,\|\, V_k) + (\{0\} \,\|\, W_{k-1}) = \{\tfrac{k+1}{2}\}^{2^{k-1}}.$$

This equation basically says that the optimal schedule does not leave any gaps: in each round exactly the maximum number of hashes are performed to meet the lower bound for binary pebbling.

Efficient in-place implementations

Without strict performance requirements, our framework for binary pebbling allows for relatively straightforward implementations. Figure 4 is showcasing a conceptually simple implementation based on Python generators. For demonstration purposes, we are using MD5 as a 128-bit length-preserving one-way function — MD5 is readily available in Python, also no practical attacks against the one-wayness of MD5 are known to this day.

By exploiting specific properties of the optimal schedule, we will next show how to implement binary pebblers with minimal overhead. In fact, we present *in-place* hash chain reversal algorithms, where the entire state of these algorithms (apart from the hash values) is represented between rounds by a single k-bit counter *only*. Below, this is shown for Jakobsson's speed-2 pebblers; refer to [20] for further results.

We use the following terminology to describe the state of a pebbler P_k (which applies to both speed-2 pebblers and optimal pebblers). Pebbler P_k is said to be *idle* if it is in rounds $[1, 2^{k-1})$, *hashing* if it is in rounds $[2^{k-1}, 2^k]$, and *redundant* if it is in rounds $(2^k, 2^{k+1})$. An idle pebbler performs no hashes at all, while a hashing pebbler will perform at least one hash per round, except for round 2^k in which P_k outputs its y_0 value. The work for a redundant pebbler P_k is taken over by its child pebblers $P_0, \ldots, P_{k-1}$ during its last $2^k - 1$ output rounds.

The important observation is that for each round r the complete state of a pebbler P_k can be deduced quickly from the binary representation of the counter $c = 2^{k+1} - r$, which counts down how many rounds are

$U_2 = \{\frac{3}{2}\}, U_k = U_{k-1} + \frac{1}{2} \,\|\, \{1\}^{2^{k-3}}$		$V_2 = \{\frac{3}{2}\}, V_k = U_{k-1} + \frac{1}{2} \,\|\, V_{k-1} + \frac{1}{2}$	
U_3	21	V_3	22
U_4	$\frac{5}{2}\frac{3}{2}11$	V_4	$\frac{5}{2}\frac{3}{2}\frac{5}{2}\frac{5}{2}$
U_5	$32\frac{3}{2}\frac{3}{2}1111$	V_5	$32\frac{3}{2}\frac{3}{2}3233$
U_6	$\frac{7}{2}\frac{5}{2}22\frac{3}{2}\frac{3}{2}\frac{3}{2}\frac{3}{2}11111111$	V_6	$\frac{7}{2}\frac{5}{2}22\frac{3}{2}\frac{3}{2}\frac{3}{2}\frac{3}{2}\frac{7}{2}\frac{5}{2}22\frac{7}{2}\frac{5}{2}\frac{7}{2}\frac{7}{2}$
U_7	$43\frac{5}{2}\frac{5}{2}2222\frac{3}{2}\frac{3}{2}\frac{3}{2}\frac{3}{2}\frac{3}{2}\frac{3}{2}\frac{3}{2}\frac{3}{2}1111111111111111$	V_7	$43\frac{5}{2}\frac{5}{2}2222\frac{3}{2}\frac{3}{2}\frac{3}{2}\frac{3}{2}\frac{3}{2}\frac{3}{2}\frac{3}{2}\frac{3}{2}43\frac{5}{2}\frac{5}{2}222243\frac{5}{2}\frac{5}{2}4344$
	avg. speed 2 speed 1		avg. speed 2 avg. speed 3

Table 1 Recursive definition of optimal schedule $T_k = \{0\}^{2^{k-1}-1} \| U_k \| V_k$ over $\frac{1}{2}\mathbb{Z}$ (no rounding). Explicit formula is in this case given by $T_k = \{0\}^{2^{k-1}-1} \,\|\, \{\frac{1}{2}(k+1-\mathrm{len}((2r) \bmod 2^{\mathrm{len}(2^k-r)}))\}_{r=2^{k-1}}^{2^k-1}$.

still left. This is illustrated in Figure 3 for a speed-2 pebbler $P_k(x)$. The pseudocode shows how to run the pebbler in-place, that is, in such a way that the storage between rounds is limited to a length-k array z of hash values and counter r. The information about the states of all pebblers running in parallel is deduced directly from c. This information includes which pebblers are present, whether these pebblers are idle or hashing, which hash values have already been computed by a pebbler, and where these are stored in array z, et cetera.

The example in Figure 3 shows the details for a P_9 pebbler at round $r = 664$. Four child pebblers P_8, P_6, P_5, P_3 are running in parallel: P_8 is hashing and has entries $z[7,8]$ in use, P_6 is idle occupying one entry $z[6]$, P_5 is hashing and has entries $z[4,5]$ in use. The P_3 pebbler has just reached its first output round occupying four entries $z[0,3]$ and outputs its y_0 value stored in $z[0]$. Subsequently, this P_3 pebbler becomes redundant and is replaced by its child pebblers P_2, P_1, P_0, which will each use one entry of array z. Entry $z[3]$ has been freed, but is immediately used again by the P_5 pebbler, which just reached the point where it starts working on its y_3 value.

The schedule for a speed-2 pebbler is integrated in the pseudocode of Figure 3. For optimal pebbling, however, we need to evaluate the formula for the optimal schedule to find the exact number of hashes to be performed by each pebbler. An intuitive way to interpret this formula is explained by means of the following example, cf. Figure 3. Consider optimal pebbler P_9 at $c = 360$ rounds from the end:

c_8	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0
1	0	1	1	0	1	0	0	0
P_8^{hashing}			P_5^{hashing}					

The formula of Table 1 for the optimal schedule (before rounding) partitions the bits of c into the two colored segments as indicated. The underlying rule is as follows. First, all the hashing pebblers are identified, ignoring the rightmost one: this results in two hashing pebblers P_8 and P_5 (idle pebbler P_6 and the rightmost hashing pebbler P_3 are ignored). Then, each of these hashing pebblers P_i gets the segment assigned starting at bit c_i and extending to the right. The number of hashes to be performed by each of these hashing pebblers — as given by the formula of the optimal schedule — exactly matches the length of these segments divided by 2. In case of P_8 this works out as $\frac{3}{2}$ hashes, and for P_5 we get $\frac{6}{2}$ hashes, hence exactly $\frac{9}{2}$ hashes are used in total for this round.

Round r:
1: output $z[0]$
2: $c \leftarrow 2^{k+1} - r$
3: $i \leftarrow \text{pop}_0(c)$
4: $z[0,i) \leftarrow z[1,i]$
5: $i \leftarrow i+1$; $c \leftarrow \lfloor c/2 \rfloor$
6: $q \leftarrow i-1$
7: **while** $c \neq 0$ **do**
8: $z[q] \leftarrow f(z[i])$
9: **if** $q \neq 0$ **then** $z[q] \leftarrow f(z[q])$
10: $i \leftarrow i + \text{pop}_0(c) + \text{pop}_1(c)$
11: $q \leftarrow i$

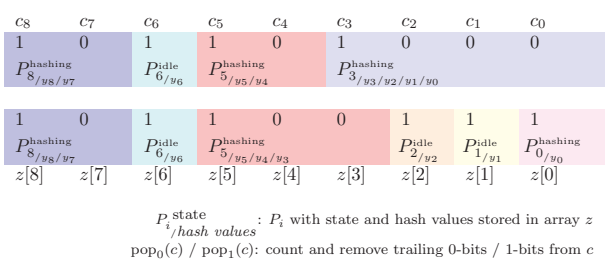


Figure 3 Pseudocode for in-place speed-2 pebbler $P_k(x)$ at output round r, $2^k < r < 2^{k+1}$. Initially, array $z[0,k)$ satisfies $z[i-1] = f^{2^k - 2^i}(x)$ for $i = 1,\ldots,k$ (left). Transition of P_9 from round $r = 664$ to $r = 665$, hence from $c = 360 = (101101000)_2$ to $c = 359 = (101100111)_2$ (right).

In general, this rule implies that no more than $\frac{k}{2}$ hashes are performed in any output round of P_k. Moreover, this simple rule will orchestrate the entire computation, ensuring that all intermediate hash values are computed right on time — not too late to fail producing an output on time, and not too early, before another free entry in array $z[0,k)$ becomes available. The optimized implementations in [20] are based on this rule.

Lower bound

Optimal binary pebbling achieves a space-time product of $0.50k^2$ for a chain of length $n = 2^k$. In an upcoming paper with Niels de Vreede, we will show how to reduce the space-time product to $0.46k^2$ by means of Fibonacci pebbling and how to reduce this even further down to just $0.37k^2$ by more intricate pebbling algorithms. We note that Coppersmith and Jakobsson [3] gave a lower bound of $0.25k^2$, but whether this bound can be attained is doubtful: the lower bound is derived without taking into account any limits on the number of hashes per output round.

Incidentally, the lower bound of $0.25k^2$ had been found already in a completely different context [7], for a similar problem studied in the area of algorithmic (or, automatic, computational) differentiation [6]. The lower bound applies to the space-time complexity of so-called *checkpointing* for the reverse (or, adjoint, backward) mode of algorithmic differentiation. In contrast to our case, however, there it is even possible to attain the lower bound [5]. The critical difference is that in the setting of algorithmic differentiation the goal is basically to minimize the *total time* for performing this task (or, equivalently, to minimize the *amortized time* per output round). This contrasts sharply with the goal in the cryptographic setting, where we want to minimize the *worst case time* per output round while performing this task.

```
import hashlib, itertools

f = lambda x: hashlib.md5(x).digest()

tR = lambda k,r: 0 if r < 2**k - 1 else 2**k - 1
t1 = lambda k,r: 1
t2 = lambda k,r: 0 if r < 2**(k-1) else 2 if r < 2**k - 1 else 1
tS = lambda k,r: 0 if r < 2**(k-1) else ((k + r) % 2 + k + 1 - ((2 * r) % (2**(2**k - r).bit_length())).bit_length()) // 2

def P(k,x):
    y = [None] * k + [x]
    i = k; g = 0
    for r in range(1, 2**k):
        for _ in range(t(k,r)):
            z = y[i]
            if g == 0: i -= 1; g = 2**i
            y[i] = f(z)
            g -= 1
        yield
    yield y[0]
    for v in itertools.zip_longest(*(P(i-1, y[i]) for i in range(1, k+1))):
        yield next(filter(None, v))

t = eval(input())
k = int(input())
x = f(b'')
for v in P(k, x):
    if v: print(v.hex())
```

Figure 4 Python program for recursive binary pebblers without any optimizations, cf. definition of $P_k(x)$ in Figure 1. Inputs: `tR`/`t1`/`t2`/`tS` for rushing/speed-1/speed-2/optimal and nonnegative integer `k`. `P(k,x)` is a Python generator: each evaluation of a `yield` expression corresponds to a round of $P_k(x)$.

References

1 J. Alwen, B. Chen, K. Pietrzak, L. Reyzin, and S. Tessaro, Scrypt is maximally memory-hard, *Advances in Cryptology–EUROCRYPT '17*, LNCS 10212, Springer, 2017, pp. 33–62.

2 J.P. Boly, A. Bosselaers, R. Cramer, R. Michelsen, S. Mjølsnes, F. Muller, T. Pedersen, B. Pfitzmann, P. de Rooij, B. Schoenmakers, M. Schunter, L. Vallée and M. Waidner, The ESPRIT project CAFE–High security digital payment systems, *Computer Security–ESORICS 94*, LNCS 875, Springer, 1994, pp. 217–230.

3 D. Coppersmith and M. Jakobsson, Almost optimal hash sequence traversal, *Financial Cryptography 2002*, LNCS 2357, Springer, 2002, 102–119.

4 P. Flajolet and A. Odlyzko, Random mapping statistics, *Advances in Cryptology–EUROCRYPT '89*, LNCS 434, Springer, 1989, pp. 329–354.

5 A. Griewank, Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation, *Optimization Methods and Software* 1(1) (1992), 35–54.

6 A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, SIAM, 2008, 2nd edition.

7 J. Grimm, L. Potter and N. Rostaing-Schmidt, Optimal time and minimum space-time product for reversing a certain class of programs, *Computational Differentiation–Techniques, Applications, and Tools*, SIAM, 1996, pp. 95–106.

8 J. Håstad and M. Näslund, Key feedback mode: a keystream generator with provable security, *First Modes of Operation Workshop*, *Baltimore, MD, October 2000*, NIST.

9 G. Hurlbert, Recent progress in graph pebbling, *Graph Theory Notes of New York* 49 (2005), 25–37.

10 G. Itkis and L. Reyzin, Forward-secure signatures with optimal signing and verifying, *Advances in Cryptology–CRYPTO '01*, LNCS 2139, Springer, 2001, pp. 332–354.

11 M. Jakobsson, Fractal hash sequence representation and traversal, *Proc. IEEE International Symposium on Information Theory (ISIT '02)*, IEEE, 2002, p. 437. Full version eprint.iacr.org/2002/001.

12 L. Lamport, Password authentication with insecure communication, *Communications of the ACM* 24(11) (1981), 770–772.

13 L. Levin, One-way function and pseudorandom generators, *Proc. 17th Symposium on Theory of Computing (STOC '85)*, ACM, 1985, pp. 363–365.

14 R. Merkle, A digital signature based on a conventional encryption function, *Advances in Cryptology–CRYPTO '87*, LNCS 293, Springer, 1987, 369–378.

15 S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system, October 31, 2008, bitcoin.org/bitcoin.pdf.

16 J. Nordström, Pebble games, proof complexity, and time-space trade-offs, *Logical Methods in Computer Science* 9(3:15) (2013), 1–63.

17 T.P. Pedersen, Electronic payments of small amounts, *Security Protocols*, LNCS 1189, Springer, 1996, pp. 59–68.

18 K. Perumalla, *Introduction to Reversible Computing*, CRC Press, 2013.

19 R.L. Rivest and A. Shamir, Payword and micromint: Two simple micropayment schemes, *Security Protocols*, LNCS 1189, Springer, 1996, pp. 69–87.

20 B. Schoenmakers, Explicit optimal binary pebbling for one-way hash chain reversal, *Financial Cryptography 2016*, LNCS 9603, Springer, 2016, pp. 299–320. Sample code in Python, Java, C at www.win.tue.nl/~berry/pebbling.

21 D.H. Yum, J.W. Seo, S. Eom and P.J. Lee, Single-layer fractal hash chain traversal with almost optimal complexity, *Topics in Cryptology–CT-RSA '09*, LNCS 5473, Springer, 2009, pp. 325–339.