""Fast"" catalytic algorithms for APSP

Nathan Sheffield

November 19 2025

1 The basic algorithm

Suppose you want to solve APSP in CL. How fast can you do it? If all weights are at most $\mathsf{polylog}(n)$ bits, the following says you can get away with time $\widetilde{O}\left(n^{6.33}\right)$. Which isn't so fast that you'd actually ever run this, but definitely better than the $n^{\odot}$ you'd get from [BCKLS14] if you weren't trying.

Theorem 1.1. There is a time-$\widetilde{O}\left(n^{4+\log_2(5)}\log(W)^2\right)$ CL algorithm[1] that, given a matrix $A \in [W]^{n\times n}$, adds the APSP matrix $A^{\star n}$ into a designated portion of the catalytic tape.

Proof. The relevant question is: if I have an algorithm P to add some matrices B and C into the catalytic tape, how many times do I have to run this P and how much additional overhead do I have to do if I want to now add $B \star C$? If we take an n^2 factor, we can look at the cost of computing a single $(\min, +)$ inner product. Which is really just the cost of computing the min of n values, since the additions are ezmoney.

Ok, so the deal is: we can toggle back and forth between looking at $\tau_1, \dots, \tau_n$ and $\tau_1 + x_1, \dots, \tau_n + x_n$. We want to, with not so much work and not too many toggles, add $\min(x_1, \dots, x_n)$ to a designated output register. This is going to be some nested version of the same indexing trick we use to evaluate MAJ gates. But this will be a little grosser oops, hopefully you can bear with me.

We're thinking of the catalytic tape as broken up into registers, each holding values mod $R = 100W^{100}n^{100}$. We have input registers $\mathsf{INPUT}_1, \dots, \mathsf{INPUT}_n$ into which we have the ability to add $x_1, \dots, x_n$, and we have an output register OUTPUT_n into which we would like to add $\min(x_1, \dots, x_n)$. Let us also define the following helper registers:

- For every $i \in [n]$, allocate a register KEY_i, and R further registers $\mathsf{ARRAY}_i[1], \dots, \mathsf{ARRAY}_i[R]$.
- For every pair $(i, j) \in [n] \times [n]$, allocate registers $\mathsf{KEY}_{i,j}$ and $\mathsf{ARRAY}_{i,j}[1], \dots, \mathsf{ARRAY}_{i,j}[R]$.

We'll define the algorithm piece-by-piece; if you're a crazy person who just wants to see the whole pseudocode I guess you can instead look at Algorithm 5. But for the rest of us, let's start by noting that we can toggle $(x_i - x_j)$ into all of the $\mathsf{KEY}_{i,j}$'s if we so desire:

Algorithm 1 Adding $(x_i - x_j)$ into each $\mathsf{KEY}_{i,j}$

1: $\forall i, j,\ \mathsf{KEY}_{i,j} \leftarrow \mathsf{KEY}_{i,j} - (\mathsf{INPUT}_i - \mathsf{INPUT}_j)$
2: $\forall i,\ \mathsf{INPUT}_i \leftarrow \mathsf{INPUT}_i + x_i$ ▷ Here we use P
3: $\forall i, j,\ \mathsf{KEY}_{i,j} \leftarrow \mathsf{KEY}_{i,j} + (\mathsf{INPUT}_i - \mathsf{INPUT}_j)$

Now, we will show an algorithm with the following property: it adds some small (i.e. smaller than R) value to each KEY_i, except for the i corresponding to the minimum[2] x_i, to which it adds 0. First, here's a version of that procedure that's not especially time-efficient:

[1] Here I'm using a relaxed definition of CL where you have RAM access to the tape, you get randomness, and you're even allowed to fail to reset the tape with small probability over that randomness. Randomness isn't needed if $W \leq \mathsf{poly}(n)$.

[2] Here we assume there is a *unique* minimum. Since this may in fact not be the case, we should instead tack on the index i as lower-order bits for each x_i to break ties. This increases the weight sizes by only a factor of n, so is totally fine for our runtime.

Algorithm 2 A first attempt at determining the minimum x_i

1: $\forall i,\ \mathsf{KEY}_i \leftarrow \mathsf{KEY}_i - \sum_j \mathsf{ARRAY}_{i,j}[\mathsf{KEY}_{i,j}]$
2: $\forall i, j,\ \mathsf{KEY}_{i,j} \leftarrow \mathsf{KEY}_{i,j} + (x_i - x_j)$ ▷ Here we use Algorithm 1
3: **for all** $0 < \ell \leq W$ **do**
4: $\quad \forall i, j,\ \mathsf{ARRAY}_{i,j}[\mathsf{KEY}_{i,j} - \ell] \leftarrow \mathsf{ARRAY}_{i,j}[\mathsf{KEY}_{i,j} - \ell] + 1$
5: **end for**
6: $\forall i, j,\ \mathsf{KEY}_{i,j} \leftarrow \mathsf{KEY}_{i,j} - (x_i - x_j)$ ▷ Here we use Algorithm 1
7: $\forall i,\ \mathsf{KEY}_i \leftarrow \mathsf{KEY}_i + \sum_j \mathsf{ARRAY}_{i,j}[\mathsf{KEY}_{i,j}]$

To see why this works, observe that if $x_i \leq x_j$ for all j, then we will always have $(x_i - x_j) \in [-W, \ldots, 0]$. So, if the initial value of $\mathsf{KEY}_{i,j}$ is τ, the array indices modified by the for loop all lie in $[\tau - 2W, \ldots, \tau - 1]$. Since $R > 2W$, this ensures that the value we add to KEY_i on line 7 is the same as the value we subtracted on line 1. On the other hand, if $x_i > x_j$ for any j, then we have $(x_i - x_j) \in [1, \ldots, W]$, so the for loop will add 1 to $\mathsf{ARRAY}_{i,j}[\tau]$, which will ensure that the value added on line 7 is different than the value added on line 1. (Note that the difference can be at most $n < R$, so this will not be masked by modular overflow.)

The issue with Algorithm 2 is the fact that our runtime is linear in W — even if all of our weights are only $O(\log(n))$ bits, this could still be an enormous factor. So here is a better approach:

Algorithm 3 A better algorithm for determining the minimum x_i

1: **for all** i, j **and all** $0 \leq \ell \leq \lceil \log(R) \rceil$ **do**
2: $\quad$ **if** there exists a multiple of 2^ℓ in $[\mathsf{KEY}_{i,j} - W, \ldots, \mathsf{KEY}_{i,j} - 1]$ **then**
3: $\quad\quad$ Let y be the last such multiple
4: $\quad\quad \mathsf{KEY}_i \leftarrow \mathsf{KEY}_i - \mathsf{ARRAY}_{i,j}[y]$
5: $\quad$ **end if**
6: **end for**
7:
8: $\forall i, j,\ \mathsf{KEY}_{i,j} \leftarrow \mathsf{KEY}_{i,j} + (x_i - x_j)$ ▷ Here we use Algorithm 1
9: **for all** i, j **and all** $0 \leq \ell \leq \lceil \log(R) \rceil$ **do**
10: $\quad$ **if** there exists a multiple of 2^ℓ in $[\mathsf{KEY}_{i,j} - W, \ldots, \mathsf{KEY}_{i,j} - 1]$ **then**
11: $\quad\quad$ Let z be the first such multiple
12: $\quad\quad \mathsf{ARRAY}_{i,j}[z] \leftarrow \mathsf{ARRAY}_{i,j}[z] + 1$
13: $\quad$ **end if**
14: **end for**
15: $\forall i, j,\ \mathsf{KEY}_{i,j} \leftarrow \mathsf{KEY}_{i,j} - (x_i - x_j)$ ▷ Here we use Algorithm 1
16:
17: **for all** i, j **and all** $0 \leq \ell \leq \lceil \log(R) \rceil$ **do**
18: $\quad$ **if** there exists a multiple of 2^ℓ in $[\mathsf{KEY}_{i,j} - W, \ldots, \mathsf{KEY}_{i,j} - 1]$ **then**
19: $\quad\quad$ Let y be the last such multiple
20: $\quad\quad \mathsf{KEY}_i \leftarrow \mathsf{KEY}_i + \mathsf{ARRAY}_{i,j}[y]$
21: $\quad$ **end if**
22: **end for**

This indeed requires much less time: now we're only doing $O(\log(W))$ register operations for each i, j, as opposed to $\Omega(W)$. To see correctness, observe once again that if $(x_i - x_j) \leq 0$ then the two intervals we consider do not overlap, so the value added on line 20 is the same as the values subtracted on line 4. However, if $(x_i - x_j) \in [1, \ldots, W]$, then the interval I considered on lines 2 and 18 overlaps with the interval I' considered on line 10. Consider the largest ℓ such that there is a multiple of 2^ℓ in $I \cap I'$. The last multiple of 2^ℓ in I must be the same as the first multiple of 2^ℓ in I', since if there were a later or earlier multiple of 2^ℓ in $I \cap I'$ then it would also be a multiple of $2^{\ell+1}$ in their intersection. So, at least one of the values considered on line 20 must have been incremented by 1 from its value on line 4. Since we still consider only at most $n \cdot \lceil \log(W) \rceil < R$ values for each KEY_i, it is still not possible for this value to sum up to 0 mod R. So, we have added some nonzero value to each KEY_i except for the i corresponding to the minimum x_i.

Alright, we've now computed something kind of like an indicator for the argmin of x_i. It remains to actually compute x_i. This we can do using another layer of exactly the same idea as Algorithm 2.

Algorithm 4 Full program for minimum

1: $\mathsf{OUT} \leftarrow \mathsf{OUT} - \sum_i \mathsf{ARRAY}_i[\mathsf{KEY}_i]$
2: $\forall i,\ \mathsf{KEY}_i \leftarrow \begin{cases} \mathsf{KEY}_i \text{ if } x_i \text{ is the minimum} \\ \mathsf{KEY}_i + \text{ something else otherwise} \end{cases}$ ▷ Here we use Algorithm 3
3:
4: $\forall i,\ \mathsf{ARRAY}_i[\mathsf{KEY}_i] \leftarrow \mathsf{ARRAY}_i[\mathsf{KEY}_i] - \mathsf{INPUT}_i$
5: $\forall i,\ \mathsf{INPUT}_i \leftarrow \mathsf{INPUT}_i + x_i$ ▷ Here we use P
6: $\forall i,\ \mathsf{ARRAY}_i[\mathsf{KEY}_i] \leftarrow \mathsf{ARRAY}_i[\mathsf{KEY}_i] + \mathsf{INPUT}_i$
7:
8: $\forall i,\ \mathsf{KEY}_i \leftarrow \begin{cases} \mathsf{KEY}_i \text{ if } x_i \text{ is the minimum} \\ \mathsf{KEY}_i - \text{ something else otherwise} \end{cases}$ ▷ Here we use Algorithm 3
9: $\mathsf{OUT} \leftarrow \mathsf{OUT} + \sum_i \mathsf{ARRAY}_i[\mathsf{KEY}_i]$

Here, if x_i is the minimum (and not otherwise), then KEY_i will be the same on lines 4-6 as it is on lines 1 and 9. Lines 4-6 result in $\mathsf{ARRAY}_i[\mathsf{KEY}_i]$ being increased by x_i, so this overall program will result in x_i being added to the output. Also, note that all operations done are reversible so this does indeed yield a catalytic routine.

This program requires 5 calls to P: 2 for each iteration of Algorithm 3, and 1 additional one on line 5. The work performed in addition to that consists of $O(n^2 \log(W))$ many register operations. Each of these register operations either consists of adding the value of some register into another, where possibly the indices of these registers are themselves determined by the value of some KEY_i or $\mathsf{KEY}_{i,j}$. If $W \leq \mathsf{poly}(n)$, this is easy to handle, since we can look up the relevant values and do this computation in only $\mathsf{polylog}(n)$ time in our workspace. If, however, W is much larger, then things become slightly trickier. If we knew the indices of the pair of registers we wanted to add, we could perform this addition easily in $\widetilde{O}(\log(W))$ time by the grade-school method. But the issue is that these indices don't even fit into our workspace — we're trying to index into a catalytic memory of size *larger than* $\mathsf{poly}(n)$.

The solution is to hash. We only ever look at $\mathsf{poly}(n)$ many addresses of the catalytic memory over the course of the whole algorithm. So, if we choose a sufficiently large random $O(\log(n) \log\log(W))$-bit prime p, it's very likely that no pair of those addresses take the same value mod p. Which means that our algorithm will whp behave identically if, every time we would look up an address, we instead compute the remainder of that address mod p (which we can do using our workspace and $\widetilde{O}(\log(W))$ time), and look up that "aliased" address instead.

In order to compute the nth $(\min, +)$-power, we'll need to recurse this process for $\log_2(n)$ levels. So, overall the cost of the algorithm is $\widetilde{O}\left(n^2 \cdot n^2 \log(W) \cdot \log(W) \cdot 5^{\log_2(n)}\right) = \widetilde{O}\left(n^{4+\log_2(5)} \log(W)^2\right)$. ■

2 Some further improvements

For APSP you want the nth $(\min, +)$ power. But if you just want to know distances within a small number of hops, you could compute a smaller $(\min, +)$ power — that is, $A^{\star k}$ for some $k < n$. The exact same argument from Theorem 1.1 will let you compute this in time $\widetilde{O}\left(n^4 k^{\log_2(5)} \log(W)^2\right)$. But if $k \ll n$, you can do better than that algorithm by exploiting depth/work tradeoffs for computing the min.

Proposition 2.1. For any integer q and any $k \leq n^{\left(\frac{1}{q^2 \log_2(5)}\right)}$, there is a time-$\widetilde{O}\left(n^{3+2/q} \log(W)^2\right)$ CL algorithm that, given a matrix $A \in [W]^{n \times n}$, adds the $(\min, +)$-power $A^{\star k}$ into a designated portion of the catalytic tape.

Proof. Instead of computing the min as in that algorithm, you can do it as a depth-q tree of mins of $n^{1/q}$ many registers. This costs you a factor of q in recursive depth, but saves you a factor of $n^{1-1/q}$ in time for each recursive call. So, you get a runtime of $\widetilde{O}\left(n^{3+1/q}k^{q\log_2(5)}\log(W)^2\right) \leq \widetilde{O}\left(n^{3+2/q}\log(W)^2\right)$. ■

One note is that if $k = n^{o(1)}$ and $W = 2^{o(n)}$ then this gets you the (conjecturally) optimal runtime of $n^{3+o(1)}$. But actually, even if you only care about APSP as opposed to this k-hop version, you can use these ideas to speed up Theorem 1.1. The point is: the lower levels of recursion get run many many times, while the upper levels do not. So we can vary how aggressively we're blowing up this min tree depending on where in the recursion we are — for the first few repeated squarings we'll want to unfold it a lot because paying $n^4\log(W)^2$ each time is going to bite us, but for the last few we can afford the $n^4\log(W)^2$ in exchange for having to call the lower recursive levels fewer times. If you try to optimize this, I think you end up with a pretty weird runtime: $n^{2+\log_2(5)+\pi^2/6} \approx n^{5.97}$. lol. Here's a relevant lemma:

Lemma 2.2. For any integer N, and any $c \geq 1$, there exist integers $q_1, \ldots, q_N$ such that

$$\max_{i\in[N]}\left(\frac{N}{q_i} + c\sum_{j\leq i} q_j\right) \leq \left(c + \frac{\pi^2}{6} - 1\right)N.$$

Proof. We will use the following assignment:

$$q_i = \begin{cases} 1 & \text{if } i \leq \left(1 + \frac{\pi^2}{6c} - \frac{2}{c}\right)N \\ v & \text{if } N\left(1 - \sum_{\ell=v}^{\infty}\frac{1}{c\ell^2(\ell-1)}\right) \leq i \leq N\left(1 - \sum_{\ell=v+1}^{\infty}\frac{1}{c\ell^2(\ell-1)}\right). \end{cases}$$

First, we've got to verify that this even gives a value for every q_i. That is, that every i with $\left(1 + \frac{\pi^2}{6c} - \frac{2}{c}\right)N < i < N$ belongs to the interval $\left[N\left(1 - \sum_{\ell=v}^{\infty}\frac{1}{c\ell^2(\ell-1)}\right), \ldots, N\left(1 - \sum_{\ell=v+1}^{\infty}\frac{1}{c\ell^2(\ell-1)}\right)\right]$ for some v. First, note that for large v the upper endpoints of these intervals get arbitrarily close to N. So, it remains to show that $N\left(1 - \sum_{\ell=2}^{\infty}\frac{1}{c\ell^2(\ell-1)}\right) = \left(1 + \frac{\pi^2}{6c} - \frac{2}{c}\right)N$. This follows from the Basel series:

$$\begin{aligned} 1 - \sum_{\ell=2}^{\infty}\frac{1}{c\ell^2(\ell-1)} &= 1 - \frac{1}{c}\left(\sum_{\ell=2}^{\infty}\frac{1}{\ell-1} - \frac{1}{\ell} - \frac{1}{\ell^2}\right) \\ &= 1 + \frac{1}{c}\left(\sum_{\ell=1}^{\infty}\frac{1}{\ell^2}\right) - \frac{1}{c} - \frac{1}{c}\left(\sum_{\ell=2}^{\infty}\frac{1}{\ell-1} - \frac{1}{\ell}\right) \\ &= 1 + \frac{\pi^2}{6c} - \frac{2}{c}. \end{aligned}$$

Second, we have to verify that the constraint is actually satisfied for all i. When $q_i = 1$, this is immediate since

$$N + c\sum_{j\leq i} q_j = N + ci \leq N + \left(c + \frac{\pi^2}{6} - 2\right)N.$$

On the other hand, when $q_i = v$ for $v > 1$ we have

$$\begin{aligned} \frac{N}{q_i} + c\sum_{j\leq i} q_j &\leq \frac{N}{v} + \left(c + \frac{\pi^2}{6} - 2\right)N + c\sum_{2\leq k\leq v} k\left(N\left(1 - \sum_{\ell=k+1}^{\infty}\frac{1}{c\ell^2(\ell-1)}\right) - N\left(1 - \sum_{\ell=k}^{\infty}\frac{1}{c\ell^2(\ell-1)}\right)\right) \\ &= \frac{N}{v} + \left(c + \frac{\pi^2}{6} - 2\right)N + N\sum_{2\leq k\leq v} k\left(\frac{1}{k^2(k-1)}\right) \\ &= N\left(c + \frac{\pi^2}{6} - 2 + \frac{1}{v} + \left(\sum_{2\leq k\leq v}\frac{1}{k-1} - \frac{1}{k}\right)\right) \\ &= N\left(c + \frac{\pi^2}{6} - 1\right). \end{aligned}$$

■

And now the APSP algorithm follows:

Theorem 2.3. There is a time-$\widetilde{O}\left(n^{\left(2+\log_2(5)+\frac{\pi^2}{6}\right)}\log(W)^2\right)$ CL algorithm that, given a matrix $A \in [W]^{n\times n}$, adds the APSP matrix $A^{\star n}$ into a designated portion of the catalytic tape.

Proof. Let $N = \log(n)$, $c = \log_2(5)$, and $q_1, \ldots, q_N$ be as in Lemma 2.2. Follow the same algorithm as in Theorem 1.1, but when computing $A^{2^{N-\ell}} \star A^{2^{N-\ell}}$, blow up the min into q_ℓ levels. Aside from recursive calls, this product takes overhead $\widetilde{O}\left(n^{3+1/q_\ell}\log(W)^2\right)$. Over all recursive calls from higher levels, this product will be computed a total of $\prod_{i\le\ell} 5^{q_i} = 2^{\left(\sum_{i\le\ell} q_i \log_2(5)\right)}$ times. So, the total work done is

$$\begin{aligned}\sum_{\ell\in[N]} \widetilde{O}\left(n^{3+1/q_\ell}\log(W)^2\right)\cdot 2^{\left(\sum_{i\le\ell} q_i\log_2(5)\right)} &= \widetilde{O}\left(n^3\log(W)^2\right)\cdot\sum_{\ell\in[N]} 2^{\left(N/q_\ell+\log_2(5)\sum_{i\le\ell} q_i\right)}\\ &\le \widetilde{O}\left(n^3\log(W)^2\right)\cdot N\cdot 2^{\left(\log_2(5)+\frac{\pi^2}{6}-1\right)Ns}\\ &= \widetilde{O}\left(n^{\left(2+\log_2(5)+\frac{\pi^2}{6}\right)}\log(W)^2\right).\end{aligned}$$

■

Can anyone break the famous $n^{\left(\log_2(20)+\frac{\pi^2}{6}\right)}$ barrier for catalytic APSP? Only time will tell...

A Expanded pseudocode for Theorem 1.1

Algorithm 5 Unrolled version of Algorithm 4 in case you want it for some reason

$\mathsf{OUT} \leftarrow \mathsf{OUT} - \sum_i \mathsf{ARRAY}_i[\mathsf{KEY}_i]$
for all i, j **and all** $0 \le \ell \le \lceil \log(R) \rceil$ **do**
  if there exists a multiple of 2^ℓ in $[\mathsf{KEY}_{i,j} - W, \ldots, \mathsf{KEY}_{i,j} - 1]$ **then**
    Letting y be the last such multiple, $\mathsf{KEY}_i \leftarrow \mathsf{KEY}_i - \mathsf{ARRAY}_{i,j}[y]$
  end if
end for
$\forall i, j,\ \mathsf{KEY}_{i,j} \leftarrow \mathsf{KEY}_{i,j} - (\mathsf{INPUT}_i - \mathsf{INPUT}_j)$
$\forall i,\ \mathsf{INPUT}_i \leftarrow \mathsf{INPUT}_i + x_i$ ▷ Here we use P
$\forall i, j,\ \mathsf{KEY}_{i,j} \leftarrow \mathsf{KEY}_{i,j} + (\mathsf{INPUT}_i - \mathsf{INPUT}_j)$
for all i, j **and all** $0 \le \ell \le \lceil \log(R) \rceil$ **do**
  if there exists a multiple of 2^ℓ in $[\mathsf{KEY}_{i,j} - W, \ldots, \mathsf{KEY}_{i,j} - 1]$ **then**
    Letting z be the first such multiple, $\mathsf{ARRAY}_{i,j}[z] \leftarrow \mathsf{ARRAY}_{i,j}[z] + 1$
  end if
end for
$\forall i, j,\ \mathsf{KEY}_{i,j} \leftarrow \mathsf{KEY}_{i,j} + (\mathsf{INPUT}_i - \mathsf{INPUT}_j)$
$\forall i,\ \mathsf{INPUT}_i \leftarrow \mathsf{INPUT}_i + x_i$ ▷ Here we use P
$\forall i, j,\ \mathsf{KEY}_{i,j} \leftarrow \mathsf{KEY}_{i,j} - (\mathsf{INPUT}_i - \mathsf{INPUT}_j)$
for all i, j **and all** $0 \le \ell \le \lceil \log(R) \rceil$ **do**
  if there exists a multiple of 2^ℓ in $[\mathsf{KEY}_{i,j} - W, \ldots, \mathsf{KEY}_{i,j} - 1]$ **then**
    Letting y be the last such multiple, $\mathsf{KEY}_i \leftarrow \mathsf{KEY}_i + \mathsf{ARRAY}_{i,j}[y]$
  end if
end for
$\forall i,\ \mathsf{ARRAY}_i[\mathsf{KEY}_i] \leftarrow \mathsf{ARRAY}_i[\mathsf{KEY}_i] - \mathsf{INPUT}_i$
$\forall i,\ \mathsf{INPUT}_i \leftarrow \mathsf{INPUT}_i + x_i$ ▷ Here we use P
$\forall i,\ \mathsf{ARRAY}_i[\mathsf{KEY}_i] \leftarrow \mathsf{ARRAY}_i[\mathsf{KEY}_i] + \mathsf{INPUT}_i$
for all i, j **and all** $0 \le \ell \le \lceil \log(R) \rceil$ **do**
  if there exists a multiple of 2^ℓ in $[\mathsf{KEY}_{i,j} - W, \ldots, \mathsf{KEY}_{i,j} - 1]$ **then**
    Letting y be the last such multiple, $\mathsf{KEY}_i \leftarrow \mathsf{KEY}_i + \mathsf{ARRAY}_{i,j}[y]$
  end if
end for
$\forall i, j,\ \mathsf{KEY}_{i,j} \leftarrow \mathsf{KEY}_{i,j} - (\mathsf{INPUT}_i - \mathsf{INPUT}_j)$
$\forall i,\ \mathsf{INPUT}_i \leftarrow \mathsf{INPUT}_i + x_i$ ▷ Here we use P
$\forall i, j,\ \mathsf{KEY}_{i,j} \leftarrow \mathsf{KEY}_{i,j} + (\mathsf{INPUT}_i - \mathsf{INPUT}_j)$
for all i, j **and all** $0 \le \ell \le \lceil \log(R) \rceil$ **do**
  if there exists a multiple of 2^ℓ in $[\mathsf{KEY}_{i,j} - W, \ldots, \mathsf{KEY}_{i,j} - 1]$ **then**
    Letting z be the first such multiple, $\mathsf{ARRAY}_{i,j}[z] \leftarrow \mathsf{ARRAY}_{i,j}[z] + 1$
  end if
end for
$\forall i, j,\ \mathsf{KEY}_{i,j} \leftarrow \mathsf{KEY}_{i,j} + (\mathsf{INPUT}_i - \mathsf{INPUT}_j)$
$\forall i,\ \mathsf{INPUT}_i \leftarrow \mathsf{INPUT}_i + x_i$ ▷ Here we use P
$\forall i, j,\ \mathsf{KEY}_{i,j} \leftarrow \mathsf{KEY}_{i,j} - (\mathsf{INPUT}_i - \mathsf{INPUT}_j)$
for all i, j **and all** $0 \le \ell \le \lceil \log(R) \rceil$ **do**
  if there exists a multiple of 2^ℓ in $[\mathsf{KEY}_{i,j} - W, \ldots, \mathsf{KEY}_{i,j} - 1]$ **then**
    Letting y be the last such multiple, $\mathsf{KEY}_i \leftarrow \mathsf{KEY}_i - \mathsf{ARRAY}_{i,j}[y]$
  end if
end for
$\mathsf{OUT} \leftarrow \mathsf{OUT} + \sum_i \mathsf{ARRAY}_i[\mathsf{KEY}_i]$