Catalytic Approaches to the Tree Evaluation Problem

James Cook
University of Toronto
Canada
jcook@cs.berkeley.edu

Ian Mertz
University of Toronto
Canada
mertz@cs.toronto.edu

ABSTRACT

The study of branching programs for the Tree Evaluation Problem (TreeEval), introduced by S. Cook et al. (TOCT 2012), remains one of the most promising approaches to separating L from P. Given a label in $[k]$ at each leaf of a complete binary tree and an explicit function in $[k]^2 \to [k]$ for recursively computing the value of each internal node from its children, the problem is to compute the value at the root node. (While the original problem allows an arbitrary-degree tree, we focus on binary trees.) The problem is parameterized by the alphabet size k and the height h of the tree. A branching program implementing the straightforward recursive algorithm uses $\Theta((k+1)^h)$ states, organized into $2^h - 1$ layers of width up to k^h. Until now no better deterministic algorithm was known.

We present a series of three new algorithms solving TreeEval. They are inspired by the work of Buhrman et al. on *catalytic space* (STOC 2012), applied outside the catalytic-space setting. First we give a novel branching program with $2^{4h}\operatorname{poly}(k)$ layers of width 2^{3k}, which beats the straightforward algorithm when $h = \omega(k/\log k)$. Next we give a branching program with $k^{2h}\operatorname{poly}(k)$ layers of width k^3. This has total size comparable to the straightforward algorithm, but is implemented using the catalytic framework. Finally we interpolate between the two algorithms to give a branching program with $(O(\frac{k}{h}))^{2h}\operatorname{poly}(k)$ layers of width $(O(\frac{k}{h}))^{\epsilon h}$ for any constant $\epsilon > 0$, which beats the straightforward algorithm for all $h \geq k^{1/2+\operatorname{poly}\epsilon}$. These are the first deterministic branching programs to beat the straightforward algorithm, but more importantly this is the first non-trivial approach to proving deterministic upper bounds for TreeEval.

We also contribute new machinery to the catalytic computing program, which may be of independent interest to some readers.

CCS CONCEPTS

• **Theory of computation → Computational complexity and cryptography**; *Design and analysis of algorithms.*

KEYWORDS

complexity theory, branching programs, catalytic computing, tree evaluation problem

ACM Reference Format:
James Cook and Ian Mertz. 2020. Catalytic Approaches to the Tree Evaluation Problem. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC '20), June 22–26, 2020, Chicago, IL, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3357713.3384316

1 INTRODUCTION

The deterministic time and space classes, such as L, P, PSPACE, EXP, and EXPSPACE are fundamental to complexity theory. While the containments $\mathsf{SPACE}(k) \subseteq \mathsf{TIME}(2^k)$ and $\mathsf{TIME}(k) \subseteq \mathsf{SPACE}(k)$ are exercises that would show up in a first complexity course, figuring out whether these containments are strict or not has proved to be one of the greatest challenges in the field. As an example, one way to separate P from PSPACE would be to separate P from NP, but determining whether P = NP has remained unsolved for fifty years. The Tree Evaluation Problem [7] has emerged over the past ten years as a candidate for separating L from P.

1.1 The Tree Evaluation Problem and L vs. P

Definition 1 (Tree Evaluation Problem [7])**.** The *tree evaluation problem* $\mathsf{TreeEval}_{h,k}$ is parameterized by a height h and an alphabet size k. The input is a full binary tree of height h, where every leaf is labeled with an element of $[k]$ and every internal node is labeled with a function from $[k] \times [k]$ to $[k]$. The output is the value of the root of the tree, where the tree is evaluated bottom-up in the natural way. We will often omit the subscripts and write TreeEval.

(In the original statement of the problem [7], the degree of the internal nodes in the tree is an additional parameter d. Here, we focus exclusively on binary trees ($d = 2$).)

The input to $\mathsf{TreeEval}_{h,k}$ has size $(2^{h-1}-1)k^2 \log k + 2^{h-1} \log k = O(2^h \operatorname{poly}(k))$. The problem is in P: it can be solved in polynomial time by evaluating every node, starting from the leaves, in an order that ensures a node's two children get evaluated before its parent.

However, it is not a log-space algorithm. The space used depends on the order in which the nodes are evaluated, since child values can be forgotten once a parent is evaluated. An argument based on a "pebbling game" [7, 14] shows that even the most space-efficient version of the algorithm must at some point simultaneously store h values, requiring space $\Omega(h \log k) \subseteq \omega(h + \log k)$ for non-constant h, k.

We call this algorithm the *pebbling algorithm.* (Specifically, the most efficient version of the algorithm, using $\Theta(h \log k)$ space.)

1.2 Branching Programs and Lower Bounds

In order to prove space lower bounds for TreeEval, a natural model which previous work has focused on is the *branching program* model, where space is represented by the (logarithm of the) size of the program. The pebbling algorithm described in the previous section can be translated into a branching program whose states are arranged into $2^h - 1$ layers, corresponding to the order in which the nodes are evaluated. The number of states in a layer varies depending on how many values the algorithm must remember at the corresponding point in its execution, but the pebbling-based lower bound shows that at least one layer will always have at least k^h nodes. We say this algorithm has *length* $2^h - 1$ and *width* k^h. The *size* of a branching program is the number of states; a careful analysis shows this one has size $\Theta((k+1)^h)$. (This is equivalent to $\Theta(k^h)$ so long as $h = O(k)$).)

While no unconditional lower bounds are known, a tight $\Omega(k^h)$ lower bound is known for a number of natural restrictions. In the *read-once* restriction the branching program only looks at each bit of the input at most once, while in the *thrifty* restriction the branching program must read only bits corresponding to the actual evaluation of the tree may be read (so for example if the children of a node v evaluate to x and y, the branching program must not read any values of the function at v other than the value at (x, y)). The pebbling algorithm fulfills both of these conditions, but either one of them is enough to guarantee a lower bound of $\Omega(k^h)$ [9][7], and neither of these restrictions assume any other structure on the branching program such as being layered.

1.3 Catalytic Computing

The *catalytic computing* framework of [3], which came out of a fascinating line of work [1, 2] on branching programs and circuits, proposes a novel way to use space in a more efficient way when computing circuits with simple invertible operations. The idea is deceptively simple: assume that we have a small amount of clean work space but an exponentially larger amount of "catalytic space", which is free to use but is full of junk bits that have to be returned to their original configuration at the end of the computation. Since we have no assumptions on the bits in the catalytic space it would seem like it can't help us compute anything, but Buhrman et al. [3] show that if we are working with mathematical instructions that are invertible, this invertibility can help us in two ways: first, by letting us use the space in a way that can be easily reset at the end of the computation, and second, by cleverly cancelling out the "noise" that the bits in the catalytic space introduce into the computation by inverting the computation and then subtracting off the contribution of the noise.

While there has been a flurry of work [4, 5, 8, 11, 15] following the definition of catalytic computing in [3] (see e.g. [12] for a survey of early results), the preliminary results of [1, 2] solved a slightly different type of problem. The catalytic computing model involves having a small clean work tape and exponentially more "catalytic space", but [1] and [2] study what can be done by constantly reusing a small (even constant size) work tape. Since we are looking to rule out logspace algorithms for TreeEval, it is this latter approach which seems more immediately applicable.

1.4 Our Results

In this work we show how the catalytic computing framework can be applied to the tree evaluation problem to give novel algorithms, even for the simple model of layered branching programs. We present the following three programs (recall that the layered branching program for the pebbling algorithm has length $2^h \operatorname{poly}(k)$ and width k^h)

Theorem 1 (One-hot algorithm). *There exists a layered branching program solving* $\mathsf{TreeEval}_{h,k}$ *with length at most* $4^h \operatorname{poly}(k)$ *and width* 2^{3k}.

Theorem 2 (Binary algorithm). *There exists a layered branching program solving* $\mathsf{TreeEval}_{h,k}$ *with length at most* $(2k)^{2h} \operatorname{poly}(k)$ *and width* k^3.

Theorem 3 (Hybrid algorithm). *For every* $\epsilon > 0$ *there exists a* $C = O(1/\epsilon)$ *and a branching program solving* $\mathsf{TreeEval}_{h,k}$ *with length at most* $(C\frac{k}{h} + 1)^{2h} \operatorname{poly}(k)$ *and width* $(C\frac{k}{h} + 1)^{\epsilon h}$.

While the constants in the exponent mean these algorithms don't beat the pebbling algorithm in all cases, for h large (but still $o(k)$) we do indeed achieve an $o(k^h)$ size branching program.

1.5 Important Ideas

This is the first non-trivial approach to proving upper bounds for TreeEval, and in our opinion it highlights a number of interesting ideas in catalytic computing and branching programs, which we will highlight before going into the body of the paper.

Pebbling games. The previous best-known algorithm for TreeEval was based on a strategy for the *pebbling game* on a complete binary tree.

The optimal strategy for this game is well-understood, leading to a $\Omega(k^h)$ lower bound on the number of states used by any pebbling-based branching program. Every subsequent deterministic lower bound for TreeEval, for generalized classes of branching program beyond pebbling, has arrived at the same quantity $\Omega(k^h)$, effectively by showing how to relate every algorithm back to the pebbling game. From its initial definition in [7] it has been widely believed that pebbling gives the optimal lower bound [13].

Our algorithms defeat this common lower bound by using techniques far removed from pebbling.

Use of algebraic techniques. The main result of [3] is to use catalytic computing to efficiently compute the majority of poly n bits, which they do in an algebraic way by summing the input and then using Fermat's Little Theorem. This relies on an inherent way of turning majority into an algebraic object [16]. In TreeEval, each node is labeled with an arbitrary $[k] \times [k] \to [k]$ function. The encoding we present for each algorithm determines how this function can be interpreted as an algebraic object — for example, the one-hot encoding in §3 allows us to interpret it as a sum over products corresponding to a DNF with at most one AND evaluating to 1. We then apply techniques from [3]—as well as many novel improvements on them—to get our results.

Read-once/thrifty restrictions. Our algorithms avoid the lower bounds in the read-once and thrifty models (§1.2). How do they do this? Simply put, the "catalytic space" approach involves recomputing nodes many many times and checking the evaluation of every possible pair of inputs at every node, which leads them to break the read-once and thrifty restrictions in spectacular fashion. This is the first approach to TreeEval that breaks both restrictions, and furthermore in our opinion it does so in a very natural way.

Space-bounded models. We are working in the model of small total workspace, which in some ways puts our work morally closer to the results of [1, 2] than to [3]. However at the core of our results are extensions of techniques from [3], which gives us new ways to use these techniques designed for the large (catalytic) space regime in the setting where not even the catalytic tape is available. It would be interesting to see how many of our known catalytic techniques can be carried over in this way.

Future improvements. The extensions we prove are not known to be optimal in terms of how much recomputation is needed, which is the only bottleneck in the strength of our results. Thus improving them provides a direct approach to improving our results for TreeEval. In fact, an "optimal generalization" of the lemma in question would give a logspace algorithm for TreeEval, firmly shutting the door on the approach of [7].

2 PRELIMINARIES

While we think of the input to $\mathsf{TreeEval}_{h,k}$ as being of size $2^{h-1}\log k + (2^{h-1}-1)k^2 \log k$, for our computation model it will be easier to think of our programs as always reading a whole element of $[k]$ at once.

Definition 2 (One piece of the input)**.** In the below definitions of register programs and branching programs, each instruction or state will be allowed to read one "piece" of the input to $\mathsf{TreeEval}_{h,k}$. A piece of the input is either the value associated with a leaf, or an internal node's function evaluated at one of its k^2 possible inputs.

The input consists of $2^{h-1} + (2^{h-1}-1)k^2$ pieces, each of which is a value in $[k]$.

2.1 Branching Programs

There are different definitions of branching programs. Ours is equivalent to that of S. Cook et al. [7], restricted to the deterministic case. Each state of a branching program reads one piece of the input to $\mathsf{TreeEval}_{h,k}$ (Definition 2): either the single value associated with a leaf, or an internal node's function evaluated at one of the k^2 possible inputs.

Definition 3 (Branching program [7])**.** A deterministic branching program[1] for $\mathsf{TreeEval}_{h,k}$ consists of:

- A set of *states* V, one of which is identified as the starting state.
- A set of output states identified in V, each labelled with a value for the program to output.
- For every non-output state, a piece of the input to query (Definition 2) , and a *transition function* mapping the result of the query (in $[k]$) to the next state.

If the sequence of states induced by an input to $\mathsf{TreeEval}_{h,k}$ ends at an output state (rather than looping infinitely), the program terminates with that output. The *size* of the branching program is $|V|$.

Additionally our programs will be from a restricted model called a *layered* branching program, wherein each state $v \in V$ is associated with a *layer* t, such that if v's transition function maps it to $v' \in V$ on some query, then v' is in layer $t+1$.

[1]In this paper we only consider *uniform* branching programs, which are branching programs which can be efficiently constructed. The exact notion of uniformity we use is unimportant, except for noting that any size s branching program we construct for TreeEval can certainly be constructed uniformly in $\log s$ space.

2.2 Invertible Programs and Transparent Computation

We describe our algorithms as *register machine programs*, which are described by a set of registers each storing values in some ring R, plus a list of mathematical instructions on updating those registers. Our algorithms for TreeEval use the two-element field $R = \mathbb{F}_2$, but many of our results apply more generally. Each instruction has the form $R_x \leftarrow R_x + \prod_i u_i$ (or later, $R_x \leftarrow R_x + \sum_j \prod_i u_{i,j}$), where R_x is a register and each u_i is either a constant or a register other than R_x. These are similar to the programs used in the catalytic computing work of Buhrman et al. [3]. In particular, every instruction is reversible—the instructions $R_x \leftarrow R_x + (-1)\cdot\prod u_i$ and $R_x \leftarrow R_x + \sum_j (-1)\cdot\prod_i u_{i,j}$ respectively suffice—so we call them *invertible programs*.

We will analyze the behaviour of subroutines by comparing the register values before and after a subroutine runs. Following Buhrman et al. [3], we denote the initial value of register R_i with τ_i and say that a subroutine *transparently computes* value v into register R_i if $R_i = \tau_i + v$ when it finishes. We use similar notation for vectors of registers: $\overrightarrow{\tau_i}$ is the initial value of $\overrightarrow{R_i}$, and transparently computing a vector $\overrightarrow{v}$ means ensuring $\overrightarrow{R_i} = \overrightarrow{\tau_i} + \overrightarrow{v}$.

We depart from [3] by allowing some register indices to depend on the input. For example, if v_x denotes the value of leaf node x in the input to TreeEval, then the instruction $R_{1,v_x} \leftarrow R_{1,v_x} + 1$ increments a coordinate of $\overrightarrow{R_1}$ depending on v_x. To connect register programs to branching programs, we make one restriction of our register programs, which is that each instruction uses at most one piece of the input. We can then transform register programs into branching programs:

Lemma 4. *Suppose P is a register program consisting of $|P|$ instructions using m registers over $\mathbb{F}_2$ that transparently computes $\overrightarrow{v}$ into a vector of registers $\overrightarrow{R_1}$. Then there is a layered branching program B_P of size $1 + |P|2^m + k$ which outputs v. The states of B_P other than the starting and output states are organized into $|P|$ layers of size 2^m.*

Proof. The first layer of B_P consists of a single starting state, and the last consists of k output states. Every other layer corresponds to an instruction in P, and has a state for each of the 2^m possible register configurations. A state reads whichever piece of the input the corresponding instruction uses, and its transition function leads to the state in the following layer corresponding to the new register values.

In order to make B_P output the correct value, initialize all registers to zero by choosing $0 \in {\mathbb{F}_2}^m$ in the first layer as the starting state, and designating each state in the final layer as an output state labelled with the value of $\overrightarrow{R_1}$. ◻

3 ALGORITHM 1: ONE-HOT

We first present our *one-hot* algorithm, named after the encoding scheme it uses.

Definition 4 (One-hot encoding). Let $\vec{v_i} = \{v_{i,x}\}_{x\in[k]}$ be a vector of length k. We say that $\vec{v_i}$ stores the value $x \in [k]$ if $v_{i,x} = 1$ and $v_{i,x'} = 0$ for all $x' \neq x$.

The foundation for Algorithm 1 is a formula for the one-hot encoding of a node in terms of its childrens' encodings. In TreeEval, if p is the parent of nodes ℓ and r, then for all $x \in [k]$, the coordinate $v_{p,x} = [v_p = x]$ of p's one-hot encoding is

$$v_{p,x} = \sum_{(y,z)\in f_p^{-1}(x)} [v_{\ell,y} = 1][v_{r,z} = 1]$$

To build toward Algorithm 1, in subsection 3.1 we show how to build a reversible program that transparently computes a single product $v_\ell v_r$ given programs that compute each factor, then in subsection 3.2 we extend it to efficiently compute the entire vector $\vec{v_p}$.

3.1 Binary Catalytic Products

The key tool in Theorem 1 and all our algorithms is a modified form of Lemma 4 from [3]. We state and prove it below, using a different program than was originally presented in [3] which will be easier to generalize.

Lemma 5 (Lemma 4, [3]). *Let R_ℓ, R_r and R_p be distinct registers. Let P_ℓ be an invertible program which transparently computes v_ℓ into register R_ℓ and leaves the other registers unchanged: in other words,*

$$R_\ell = \tau_\ell + v_\ell$$

$$R_i = \tau_i \quad \forall i \neq \ell.$$

Similarly, let P_r be a program which updates

$$R_r = \tau_r + v_r$$

$$R_i = \tau_i \quad \forall i \neq r$$

Then there exists an invertible program P_p which transparently computes $v_\ell v_r$ into R_p, leaving all other registers unchanged, i.e.

$$R_p = \tau_p + v_\ell v_r$$

$$R_i = \tau_i \quad \forall i \neq p$$

P_p uses only the three registers R_v, R_ℓ, R_p (not counting any space used by the programs P_ℓ and P_r) and makes two calls to P_ℓ and P_r each, plus four basic instructions of the form $R_p \leftarrow R_p \pm R_\ell R_r$.

Proof. Program P_p performs as follows:

```
1: $P_\ell$
2: $R_p \leftarrow R_p - R_\ell R_r$        ▷ $R_p = \tau_p - \tau_\ell\tau_r - v_\ell\tau_r$
3: $P_r$
4: $R_p \leftarrow R_p + R_\ell R_r$        ▷ $R_p = \tau_p + \tau_\ell v_r + v_\ell v_r$
5: $P_\ell^{-1}$
6: $R_p \leftarrow R_p - R_\ell R_r$        ▷ $R_p = \tau_p - \tau_\ell\tau_r + v_\ell v_r$
7: $P_r^{-1}$
8: $R_p \leftarrow R_p + R_\ell R_r$        ▷ $R_p = \tau_p + v_\ell v_r$
```

While correctness is given by the inline comments, we motivate this program intuitively. At some point (specifically in step 4) we add $(\tau_\ell + f_\ell)(\tau_r + f_r)$ to R_p. This yields the terms $\tau_\ell\tau_r + f_\ell\tau_r + \tau_\ell f_r + f_\ell f_r$, where the $f_\ell f_r$ term is ultimately what we want to be added to R_p. To cancel out all other terms, we use P_ℓ, P_ℓ^{-1}, P_r, and P_r^{-1} to isolate each term in succession, noting that the only spurious term that will come up is $\tau_\ell\tau_r$. All other registers were reset because they each have one forward program and one inverse program.

The number of recursive calls and basic instructions is clear, and the program P_p can be inverted by running it in reverse order, changing all + operations to − operations and vice-versa, and switching P calls with P^{-1} calls for all recursive calls. □

3.2 Parallel Binary Catalytic Products

Our algorithm for Theorem 1 will use one-hot encodings, so we will need to adapt Lemma 5 to work with vectors of registers. Thus in place of R_p, R_ℓ, and R_r, we will instead have vectors $\vec{R_p}$, $\vec{R_\ell}$, and $\vec{R_r}$, and for convenience we treat each $R_{i,x}$ as being an element in $\mathbb{F}_2$ (so + and − are both equivalent to a bitflip).

When we execute the program P_ℓ (P_r), this will flip exactly one register in $\vec{R_\ell}$ (exactly one register in $\vec{R_r}$), corresponding to the value of node ℓ (the value of node r, respectively). Our goal will be to do the same for v, i.e. add the function vector $\vec{f_v}$ to the register vector $\vec{R_v}$, where $f_{v,i}$ will be 1 if the value of the function computed at node v is i and 0 otherwise.

The key subroutine will be a version of Lemma 5 where ℓ and r are the left and right children of p. The value of $v_{x,i}$ will be 1 if the value of the function computed at node x is i, and 0 otherwise. Note that while we specialize the following lemma to the parameters of our TreeEval instance, this can easily be adapted as a generalization of Lemma 5.

Lemma 6 (Lemma 5, parallel sum version). *Let $\vec{R_\ell}$, $\vec{R_r}$ and $\vec{R_p}$ be distinct k-dimensional vectors of registers. Let $v_{\ell,x} = 1$ iff x is the value of node ℓ, and let P_ℓ be a program which transparently computes $\vec{v_\ell}$ into register $\vec{R_\ell}$ and leaves the other registers unchanged: in other words,*

$$\vec{R_\ell} = \vec{\tau_\ell} + \vec{v_\ell}$$
$$\vec{R_i} = \vec{\tau_i} \quad \forall i \neq \ell$$

Similarly, let P_r be a program which updates

$$\vec{R_r} = \vec{\tau_r} + \vec{v_r}$$
$$\vec{R_i} = \vec{\tau_i} \quad \forall i \neq r$$

where $v_{r,x} = 1$ iff r evaluates to x. Then there exists a program P_p which updates

$$\vec{R_p} = \vec{\tau_p} + \vec{v_p}$$
$$\vec{R_i} = \vec{\tau_i} \quad \forall i \neq p.$$

P_p uses only the $3k$ registers $\vec{R_p}, \vec{R_\ell}, \vec{R_r}$ (not counting any space used by the programs P_ℓ and P_r). P_p uses two calls to P_ℓ and P_r each, plus $4k^2$ basic instructions of the form $R_{p,f_p(y,z)} \leftarrow R_{p,f_p(y,z)} \pm R_{\ell,y}R_{r,z}$, where f_p is the function associated with node p of the TreeEval *instance.*

Proof. Program P_p performs as follows (note that we retain the +/− and P/P^{-1} distinctions only to stress the similarity of this program with the one in Lemma 5):

```
1:  $P_\ell$
2:  for $x$; $(y,z)$ such that $f_p(y,z) = x$ do
3:      $R_{p,x} \leftarrow R_{p,x} - R_{\ell,y}R_{r,z}$
            ▷ $R_{p,x} = \tau_{p,x} - \sum_{(y,z)\in f_p^{-1}(x)} (\tau_{\ell,y}\tau_{r,z} + v_{\ell,y}\tau_{r,z})$
4:  end for
5:  $P_r$
6:  for $x$; $(y,z)$ such that $f_p(y,z) = x$ do
7:      $R_{p,x} \leftarrow R_{p,x} + R_{\ell,y}R_{r,z}$
            ▷ $R_{p,x} = \tau_{p,x} + \sum_{(y,z)\in f_p^{-1}(x)} (\tau_{\ell,y}v_{r,z} + v_{\ell,y}v_{r,z})$
8:  end for
9:  $P_\ell^{-1}$
10: for $x$; $(y,z)$ such that $f_p(y,z) = x$ do
11:     $R_{p,x} \leftarrow R_{p,x} - R_{\ell,y}R_{r,z}$
            ▷ $R_{p,x} = \tau_{p,x} + \sum_{(y,z)\in f_p^{-1}(x)} (-\tau_{\ell,y}\tau_{r,z} + v_{\ell,y}v_{r,z})$
12: end for
13: $P_r^{-1}$
14: for $x$; $(y,z)$ such that $f_p(y,z) = x$ do
15:     $R_{p,x} \leftarrow R_{p,x} + R_{\ell,y}R_{r,z}$
            ▷ $R_{p,x} = \tau_{p,x} + \sum_{(y,z)\in f_p^{-1}(x)} v_{\ell,y}v_{r,z}$
16: end for
```

The analysis is the same as in Lemma 5, as the instructions for each pair (y, z) can be treated separately since the only instructions are $R_{p,f_p(y,z)} \leftarrow R_{p,f_p(y,z)} \pm R_{\ell,y}R_{r,z}$. Each basic instruction from Lemma 5 is now k^2 basic instructions, exactly one for each (y, z) pair, and so all counts are as claimed. □

Proof of Theorem 1. We show by induction on h that there is an invertible program of length at most $(4^h - 2)k^2$ using $3k$ binary registers which transparently computes the one-hot encoding of the value of the root node of a $\mathsf{TreeEval}_{h,k}$ instance into one set of k registers, leaving the remaining $2k$ registers at their original values. Given such a program P, Lemma 4 shows we can turn it into a layered branching program with 4^h poly(k) layers each containing 2^{3k} states. The branching program produced by Lemma 4 outputs a one-hot encoding; by relabelling the output states with their decoded values, we can turn into a program that solves $\mathsf{TreeEval}_{h,k}$.

For the base case $h = 1$, the program only needs to read the value of the single node and flip the single register corresponding to its value, so the program has length $1 \le (4^1 - 2)k^2$. For the inductive step we are given an instance $\mathsf{TreeEval}_{h+1,k}$, and we inductively assume that there exist programs P_ℓ and P_r corresponding to the children ℓ and r of the root p, each of which computes a subinstance of height h.

By Lemma 6, from this we can build a program P_p computing the value at node p which uses $3k$ registers and consists of two calls each to P_ℓ and P_r plus $4k^2$ basic instructions. Thus the total length of the program is at most $(2+2)(4^h - 2)k^2 + 4k^2 \le (4^{h+1} - 2)k^2$ as promised. For the space usage, since the programs P_ℓ and P_r work regardless of the initial state of the $3k$ registers they use and reset everything except the target registers, we will allow both of them as well as P_p to use the *same* set of $3k$ registers, relabeling them as necessary within each program call. □

4 ALGORITHM 2: BINARY

Next is the *binary* algorithm, once again named after its encoding scheme. This algorithm never uses less space than the straightforward "pebbling" algorithm described in subsection 1.1, but it is an important step toward building the "hybrid" algorithm described in section 5. It is worth noting that while it performs slightly worse than pebbling, it does so with very low width.

This algorithm uses a more compact encoding.

Definition 5 (Binary encoding). Let $\vec{v_i} = \{v_{i,b}\}_{b\in[\log k]}$ be a vector of length $\log k$. We say that $\vec{v_i}$ stores the value $x \in [k]$ if $v_{i,b} = x_b$ for all $b \in [\log k]$, where x_b is the bth bit of x when written in binary.

Working with this encoding will require moving from the binary products used by Algorithm 1 to products of fan-in $2\log k$. Following the same structure as section 3, we first show how to compute a product of more than two scalar values (subsection 4.1), then extend it to efficiently compute the entire vector $\vec{v_p}$ (subsection 4.2).

4.1 d-ary Catalytic Products

While Lemma 6 can be thought of as a generalization of Lemma 5 to accomodate sums of binary products, our next lemma will be a generalization to products of more than two variables.

Lemma 7 (Lemma 5, d-ary version). *Let $R_0, \ldots, R_d$ be distinct registers, and let $P_1 \ldots P_d$ be invertible programs where P_i updates*

$$R_i = \tau_i + v_i$$
$$R_j = \tau_j \quad \forall j \ne i$$

Then there exists an invertible program P which updates

$$R_0 = \tau_0 + \prod_i v_i$$
$$R_j = \tau_j \quad \forall j \ne 0$$

P uses only the $d+1$ registers $R_0, \ldots, R_d$ (not counting any space used by the programs P_i) and makes at most 2^d calls to each P_i, plus 2^d basic instructions of the form $R_0 \leftarrow R_0 + c\prod_i R_i$ for values c independent of the register/function values.

Proof. We define some shorthand for the sake of presenting P_v. When we say P_S we mean apply all P_i and P_i^{-1} necessary so that $R_i = \tau_i$ for all $i \in S$ and $R_i = \tau_i + v_i$ for all $i \notin S$. In other words for all $i \in S$ such that $R_i = \tau_i + v_i$ we run P_i^{-1} and for all $i \notin S$ such that $R_i = \tau_i$ we run P_i, and leave all other registers untouched. Now program P performs as follows (with c_S left undefined):

```
1: for $S \subseteq [d]$ do
2:     $P_S$
3:     $R_0 \leftarrow R_0 + c_S \prod_{i=1}^{d} R_i$
4: end for
```

At the end of this program,

$$R_0 = \tau_0 + \sum_{S\subseteq[d]} c_S \left(\prod_{i\in S} \tau_i\right)\left(\prod_{i\notin S} \tau_i + v_i\right)$$

We will now choose the coefficients c_S to make that equal $\tau_0 + \prod_{i=1}^{d} v_i$.

Expanding the polynomial, we can rewrite it as $R_0 = \tau_0 + \sum_{S\subseteq[d]} d_S \left(\prod_{i\in S}\tau_i\right)\left(\prod_{i\notin S} v_i\right)$ where $d_S = \sum_{S'\subseteq S} c_{S'}$. Since our goal is to get $R_0 = \tau_0 + 1\cdot\prod_i v_i$, we can restate our goal as: choose coefficients c_S such that $d_\emptyset = 1$ and $d_S = 0$ for all $S \neq \emptyset$.

Since $d_\emptyset = c_\emptyset$, we start by setting $c_\emptyset = 1$. Now this determines the singleton sets $d_{\{i\}} = c_{\{i\}} + c_\emptyset = 0$, namely $c_{\{i\}} = -c_\emptyset$ for all i. We similarly set the remaining c_S values in increasing order of $S \neq \emptyset$ under the partial order $\subseteq$, using the formula:

$$c_S = -\sum_{S'\subsetneq S} c_{S'}$$

This ensures that $d_S = 0$ for $S \neq \emptyset$.

The number of recursive calls to any P_i is at most once per loop iteration for a total of at most 2^d, while there is one basic instruction per S for a total of 2^d. As before the program P can be inverted by running it in reverse order, changing all + operations to − operations and vice-versa, and switching P_i calls with P_i^{-1} calls for all recursive calls. □

It should be noted that the number of calls to each P_i can be improved from at most 2^d to exactly $2^d/d$ by having the loop over all S use a Gray code [10] for order, rather than choosing the order arbitrarily. However, what will be important for our TEP algorithm is that the loop runs exactly 2^d times.

4.2 Parallel d-ary Catalytic Products

We now prove an alternative version of Lemma 6 by replacing the one-hot encoding with the binary encoding (Definition 5). If ℓ and r are children of node p and $\vec{v_\ell}$ and $\vec{v_r}$ are the binary encodings of the values at those nodes, we can compute $\vec{v_p}$ as follows:

$$v_{p,b} = \sum_{(x,y,z)\in[k]^3} [x_b = 1][f_p(y,z) = x] \prod_{b'\in[\log k]} [v_{\ell,b'} = y_{b'}][v_{r,b'} = z_{b'}]$$

where t_b is the bth bit of t written in binary.

Calculating $v_{p,b}$ in this form requires us to compute a product of fan-in $2\log k$ (not counting the constant terms $[x_b = 1]$ and $[f_p(y,z) = x]$) and thus requires considerably heavier machinery than Lemma 6, namely Lemma 7. We also need to be able to compute each bit $v_{\ell,b}$ or $v_{r,b}$ separately in order to cover all subsets of the product $\prod_{b\in[\log k]} [v_{\ell,b} = y_b][v_{r,b} = z_b]$, so we have to formulate our recursive statement a bit differently.

Again note that this is a generalization of all our previous lemmas, and the "most general" version of Lemma 5, up to improvements in the parameters themselves. We discuss these issues in Appendix A.

Lemma 8 (Lemma 5, parallel sum + d-ary version). *Let $\vec{R_\ell}$, $\vec{R_r}$ and $\vec{R_p}$ be distinct $(\log k)$-dimensional vectors of registers. For all $T \subseteq [\log k]$ let $P_\ell(T)$ be a program which updates*

$$R_{\ell,b} = \tau_{\ell,b} + v_{\ell,b} \quad \forall b \in T$$

$$R_{i,b} = \tau_{i,b} \quad \text{when } i \neq \ell \text{ or } b \notin T$$

where $v_{\ell,b} = 1$ iff the binary encoding of the value at node ℓ has a 1 in the bth position. Let $P_r(T)$ be defined similarly. Then for every $T \subseteq [\log k]$ there exists a program $P_p(T)$ which updates

$$R_{p,b} = \tau_{p,b} + v_{p,b} \quad \forall b \in T$$

$$R_{i,b} = \tau_{i,b} \quad \text{when } i \neq p \text{ or } b \notin T$$

P_p uses only the $3\log k$ registers $\vec{R_p}, \vec{R_\ell}, \vec{R_r}$ (not counting any space used by the programs P_ℓ and P_r) and makes k^2 calls to each $P_\ell(T)$ and k^2 calls total to each $P_r(T)$ for a total of $2k^2$ recursive calls, plus $O(k^3 \log k)$ basic instructions.

Proof. We recall our key equation, which we restate as

$$v_{p,b} = \sum_{(x,y,z)\in[k]^3} [x_b = 1][f_p(y,z) = x] \prod_{b'\in[\log k]} (v_{\ell,b'} + \overline{y_{b'}})(v_{r,b'} + \overline{z_{b'}})$$

This is because the indicators $[v_{\ell,b'} = y_{b'}]$ and $[v_{r,b'} = z_{b'}]$ can be replaced by taking the negation of their XOR, which is the same as taking the negation of either one and adding them together mod 2.

Like in the proof of Lemma 7, we say that P_{S_ℓ,S_r} means apply whichever $P(S'_\ell)$ and $P(S'_r)$ is necessary so that $R_{\ell,b} = \tau_\ell$ for each $b \in S_\ell$, $R_{\ell,b} = \tau_\ell + v_{\ell,b}$ for $b \notin S_\ell$, and similarly for S_r. Now program $P_p(T)$ performs as follows (with c_{S_ℓ,S_r} left undefined):

```
1: for $S_\ell, S_r \subseteq [\log k]$ do
2:     $P_{S_\ell,S_r}$
3:     for $b \in T$; $(x,y,z)$ such that $x_b = 1 \wedge f_p(y,z) = x$ do
4:         $R_{p,b} \leftarrow R_{p,b} + c_{S_\ell,S_r} \prod_{b'\in[\log k]} (R_{\ell,b'} + \overline{y_{b'}} \cdot [b' \notin S_\ell])(R_{r,b'} + \overline{z_{b'}} \cdot [b' \notin S_r])$
5:     end for
6: end for
```

The analysis is the same as in Lemma 6 and Lemma 7, where we think of S_ℓ and S_r as being one large set together, over two disjoint parts of a universe of size $2\log k$. Again we are forced to choose $c_{\emptyset,\emptyset} = 1$ and then for all other S_ℓ, S_r which are not *both* empty

$$c_{S_\ell,S_r} = \sum_{\substack{S'_\ell\subseteq S_\ell \\ S'_r\subseteq S_r \\ (S'_\ell,S'_r)\neq(S_\ell,S_r)}} -c_{S'_\ell,S'_r}$$

Since there are $2^{\log k} = k$ possible sets S_ℓ and S_r, there are k^2 possible pairs of sets, so the number of recursive calls is as claimed. Each line of basic instructions in the loop consists of k^3 basic instructions per $b \in T \subseteq [\log k]$ the number of basic instructions is also as claimed. □

Proof of Theorem 2. We show by induction on h that there is an invertible program of length $(2k)^{2h}\,\mathrm{poly}(k)$ using $3\log k$ binary registers which transparently computes the binary encoding of the value of the root node of a $\mathsf{TreeEval}_{h,k}$ instance into one set of $\log k$ registers, leaving the remaining $2\log k$ registers at their original values. As in the proof of Theorem 1, we can use Lemma 4 to transform this into a layered branching program with $(2k)^{2h}\,\mathrm{poly}(k)$ layers each containing k^3 states.

For the base case $h = 1$, the program need only read the value of the single node and flip up to $\log k$ registers corresponding to the binary encoding of its value, so the program has length $\log k \leq 4^0\,\mathrm{poly}(k)$. For the inductive step we are given a $\mathsf{TreeEval}_{h+1,k}$ instance of height $h+1$, and we inductively assume that there exist programs $P_\ell(S)$ and $P_r(S)$ corresponding to the children ℓ and r of the root p, each of which computes any subset of bits for a subinstance of height h.

By Lemma 7 from this we can build a program $P_p := P_p([\log k])$ solving the function at p using $3\log k$ registers and which makes k^2

recursive calls to P_ℓ functions plus k^2 recursive calls to P_r functions, plus $O(k^3 \log k)$ basic instructions. The total length of the program is at most $2k^2 \cdot (2k)^{2h}\,\mathrm{poly}(k) + O(k^3 \log k) \le (2k)^{2(h+1)}\,\mathrm{poly}(k)$ as promised. For the space usage we again reuse all registers for each recursive call. □

5 ALGORITHM 3: HYBRID

Our final algorithm is the *hybrid* algorithm. As the name suggests it is a synthesis of the two previous approaches: we break our registers into blocks such that each element in $[k]$ falls into only one block (one-hot), and inside the block is identified by a binary encoding (binary).

To prove Theorem 3 we no longer need to generalize Lemma 5 further, as Lemma 8 provides the most general form we need. However as mentioned before we will generalize the encoding to a *hybrid encoding* that interpolates between the one-hot and binary encodings.

Definition 6 (Hybrid encoding). Let $a \in [\log k]$ be fixed, and let $\vec{v_i} = \{v_{i,(A,B)}\}_{(A,B)\in[a]\times[k/(2^a-1)]}$ be a vector of length $a \cdot \frac{k}{2^a-1}$. Intuitively we break $[k]$ into blocks of length $2^a - 1$ so that within a block each element gets a unique non-zero binary encoding of length a. More formally for $x \in [k]$ let $E(x) = (x \mod (2^a-1))+1$ and let $F(x) = \lceil \frac{x}{2^a-1} \rceil$. We say that $\vec{v_i}$ stores the value $x \in [k]$ if $v_{i,(A,B)} = [E(x)_A = 1 \wedge F(x) = B]$ for all $(A,B) \in [a]\times[k/(2^a-1)]$. Note that for $a = 1$ and $a = \log k$ we get k blocks of size 1 and 1 block of size $\log k$ respectively, which recovers the encodings in Definition 4 and Definition 5.

For all $(A,B) \in [a] \times [k/(2^a-1)]$ the value $v_{p,(A,B)}$ is given by

$$v_{p,(A,B)} = \sum_{(x,y,z)\in[k]^3} [E(x)_A = 1 \wedge F(x) = B][f_p(y,z) = x]\cdot \prod_{b\in[a]} [v_{\ell,(b,F(y))} = E(y)_b][v_{r,(b,F(z))} = E(z)_b]$$

Note that if the output of ℓ is not y, then all bits $v_{\ell,(b,F(y))}$ are zero, and since $E(y)$ is nonzero the term will be zeroed out (and similarly for the output of r and z).

This is a product of fan-in $2a$, and so Lemma 7 will step in to do the work. However one other important component is that in any term since (y,z) is fixed all $[v_{\ell,(b,F(y))} = E(y)_b]$ factors only read from block $F(y)$ and all $[v_{r,(b,F(z))} = E(z)_b]$ only read from block $F(z)$. Thus instead of running over all subsets of $[a]$ for each block in $[\frac{k}{2^a-1}]$ separately, we can simply run over subsets of $[a]$ and apply them to every block in $[\frac{k}{2^a-1}]$ simultaneously, for a total of 2^{2a} recursive calls for P_ℓ programs and 2^{2a} for P_r programs.

While Theorem 3 gives one setting of parameters chosen to make the total size of the branching program small, in reality this approach gives a whole family of branching programs for TreeEval, in particular subsuming the constructions in Theorem 1 and Theorem 2.

Lemma 9 (Hybrid lemma). *Let $\vec{R_\ell}, \vec{R_r}$ and $\vec{R_p}$ be distinct $(a \times \frac{k}{2^a-1})$-dimensional vectors of registers. For all $T \subseteq [a]$ let $P_\ell(T)$ be a program which updates*

$$R_{\ell,(A,B)} = \tau_{\ell,(A,B)} + v_{\ell,(A,B)} \quad \forall (A,B) \in T \times [\frac{k}{2^a-1}]$$

$$R_{i,(A,B)} = \tau_{i,(A,B)} \quad \textit{when } i \neq \ell \textit{ or } A \notin T$$

where $v_{\ell,(A,B)} = 1$ iff the hybrid encoding of the value at node ℓ has a 1 in the (A,B)th position. Define $P_r(T)$ similarly. Then for every $T \subseteq [a]$ there exists a program $P_p(T)$ which updates

$$R_{p,(A,B)} = \tau_{p,(A,B)} + v_{p,(A,B)} \quad \forall (A,B) \in T \times [\frac{k}{2^a-1}]$$

$$R_{i,(A,B)} = \tau_{i,(A,B)} \quad \textit{when } i \neq p \textit{ or } A \notin T$$

$P_p(T)$ uses only the $3 \cdot \frac{ak}{2^a-1}$ registers $\vec{R_p}, \vec{R_\ell}, \vec{R_r}$ (not counting any space used by the programs P_ℓ and P_r) and makes $O(2^{2a})$ calls to P_ℓ programs and $O(2^{2a})$ calls to P_r programs, plus $O(k^3 \log k)$ basic instructions.

Proof. We can rewrite our main equation for the hybrid algorithm as

$$v_{p,(A,B)} = \sum_{(x,y,z)\in[k]^3} [E(x)_A = 1 \wedge F(x) = B][f_p(y,z) = x]\cdot \prod_{b\in[a]} (v_{\ell,(b,F(y))} + \overline{E(y)_b})(v_{r,(b,F(z))} + \overline{E(z)_b})$$

This is the same as the equation stated before this lemma, except that as in Lemma 8, we have expressed the indicators $[v_{\ell,(b,F(y))} = E(y)_b]$ and $[v_{r,(b,F(z))} = E(z)_b]$ using negations of XORs. Now program $P_p(T)$ performs as follows (with c_{S_ℓ,S_r} left undefined):

```
1: for $S_\ell, S_r \subseteq [a]$ do
2:     $P_{S_\ell,S_r}$
3:     for $A \in T; B; (x,y,z)$ such that $E(x)_A = 1 \wedge F(x) = B \wedge f_p(y,z) = x$ do
4:         $R_{p,(A,B)} \leftarrow R_{p,(A,B)} + c_{S_\ell,S_r} \prod_{b\in[a]} (R_{\ell,(b,F(y))} + \overline{E(y)_b}[b \notin S_\ell])(R_{r,(b,F(z))} + \overline{E(z)_b}[b \notin S_r])$
5:     end for
6: end for
```

Notice that we never multiply bits $R_{\ell(b,B)}$ and $R_{\ell(b',B')}$ for $B \neq B'$, and so our program is able to run the protocol from Lemma 8 "in parallel" for every block $B \in [\frac{k}{2^a-1}]$ in the inner loop, so we only need 2^{2a} recursive calls each to P_ℓ and P_r (one per iteration of the outer loop).

To determine the values of c_{S_ℓ,S_r}, let us compute the final value of $R_{p,(A,B)}$, for an arbitrary $A \in T$ and $B \in [k/(2^a-1)]$. To do this, we expand the polynomial added on line 4 of the program, and take the sum over all iterations of the loop: that is, all $S_\ell, S_r \subseteq [a]$ and all x, y, z satisfying $E(x)_A = 1 \wedge F(x) = B \wedge f_p(y,z) = x$. This produces:

$$\begin{aligned}
&R_{p,(A,B)} \\
&= \tau_{p,(A,B)} + \sum_{\substack{S_\ell,S_r\subseteq[a]\\(x,y,z)\in[k]^3}} [E(x)_A = 1 \wedge F(x) = B \wedge f_p(y,z) = x]\cdot \\
&\quad c_{S_\ell,S_r}\Big(\prod_{i\in S_\ell} \tau_{\ell,(b,F(y))}\Big)\Big(\prod_{i\notin S_\ell} (\tau_{\ell,(b,F(y))} + v_{\ell,(b,F(y))} + \overline{E(y)_b})\Big)\cdot \\
&\quad \Big(\prod_{i\in S_r} \tau_{r,(b,F(z))}\Big)\Big(\prod_{i\notin S_r} (\tau_{r,(b,F(z))} + v_{r,(b,F(z))} + \overline{E(z)_b})\Big) \\
&= \tau_{p,(A,B)} + \sum_{(x,y,z)\in[k]^3} [E(x)_A = 1 \wedge F(x) = B \wedge f_p(y,z) = x]\cdot
\end{aligned}$$

$$\sum_{S_\ell,S_r\subseteq[a]} d_{S_\ell,S_r}\Big(\prod_{i\in S_\ell}\tau_{\ell,(b,F(y))}\Big)\Big(\prod_{i\notin S_\ell}(v_{\ell,(b,F(y))}+\overline{E(y)_b})\Big)\cdot$$
$$\Big(\prod_{i\in S_r}\tau_{r,(b,F(z))}\Big)\Big(\prod_{i\notin S_r}(v_{r,(b,F(z))}+\overline{E(z)_b})\Big)$$

where $d_{S_\ell,S_r}=\sum_{S'_\ell\subseteq S_\ell,S'_r\subseteq S_R} c_{S_\ell,S_r}$. Note that this is essentially the same structure as Lemma 8, with the only differences being the binary flag $[[E(x)_A=1\wedge F(x)=B\wedge f_p(y,z)=x]$ and the exact registers being multiplied together.

As usual, if we can ensure $d_{\emptyset,\emptyset}=1$ and all other coefficients are 0, then the part after $\tau_{p,(A,B)}$ exactly matches our formula for $v_{p,(A,B)}$, and so we have $R_{p,(A,B)}=\tau_{p,(A,B)}+v_{p,(A,B)}$ as required. We start by setting $c_{\emptyset,\emptyset}=1$, and then for all S_ℓ,S_r such that $S_\ell\neq\emptyset\vee S_r\neq\emptyset$,

$$c_{S_\ell,S_r}=\sum_{\substack{S'_\ell\subseteq S_\ell\\ S'_r\subseteq S_r\\ (S'_\ell,S'_r)\neq(S_\ell,S_r)}} -c_{S'_\ell,S'_r}$$

again treating S_ℓ and S_r as being one set over disjoint parts.

Since there are 2^a possible sets S_ℓ and S_r, there are 2^{2a} possible pairs of sets, so the number of recursive calls is as claimed. □

Proof of Theorem 3. By the same induction as in Theorem 1 and Theorem 2, we can use Lemma 9 to build a program $P_p := P_p([a]\times[\frac{k}{2^a-1}])$ which finds the value at node p using $3\cdot\frac{ak}{2^a-1}$ registers and 2^{2a} recursive calls plus $O(k^3\log k)$ basic steps. The total length of the program is at most $2^{2ah}\operatorname{poly}(k)$ and by reusing space the width is $2^{3ak/(2^a-1)}$. For any C choosing $a=\log(C\frac{k}{h}+1)$ we get that

$$2^{2ah}\operatorname{poly}(k)=2^{2h\log(Ck/h+1)}\operatorname{poly}(k)=(C\frac{k}{h}+1)^{2h}\operatorname{poly}(k)$$

$$2^{3ak/(2^a-1)}\leq 2^{(3/C)h\log(Ck/h+1)}=(C\frac{k}{h}+1)^{(3/C)h}$$

and so choosing $C=\frac{3}{\epsilon}$ completes the proof. □

6 CONCLUSION

A reasonable question to ask is if a better version of Lemma 7 might be too much to ask for. Certainly in terms of TreeEval it represents one possible path directly to proving TreeEval $\in$ L. Namely if Lemma 7 can be improved in the following ways:

- make only $O(1)$ calls to each P_i
- all rounds of calls to P_i's can be parallelized as in Lemma 8 such that only $O(1)$ rounds of calls are needed
- these rounds can be parallellized such that $2^{O(d)}$ instances sharing some set of $O(d)$ target registers can be run in the same round, with only $O(d)$ registers being used in total

then it should be possible to run Lemma 8 with length $2^{O(h)}\operatorname{poly}(k)$ and width $\operatorname{poly}(k)$, yielding a logspace algorithm. However it may be that such a lemma would have implications on L itself. As the most immediate avenue to improving our main theorems, studying the feasibility of such an algorithm is our most important open problem. We discuss one potential avenue in Appendix A.

S. Cook et al. [6] offer a prize for any algorithm which, for a fixed h, proves $\mathsf{TreeEval}_{h,k}\in O(k^{h-\epsilon})$ for any constant $\epsilon>0$. Note that if $h\geq k^{1/2+\epsilon/4}$ then

$$\begin{aligned}(C\tfrac{k}{h}+1)^{(2+\epsilon)h} &\leq ((C+1)k^{1/2-\epsilon/4})^{(2+\epsilon)h}\\ &= (C'k)^{(2+\epsilon)(1/2-\epsilon/4)h}\\ &\leq k^{(1-\epsilon^2/4+\log C'/\log k)h}\ll k^{h-\epsilon}\end{aligned}$$

and so we far surpass what is required for the prize. However to get an $o(k^h)$ upper bound for *all* h, the catalytic technique seems to inevitably require $3d$ registers for a representation of length d, and so getting more efficient algorithms for succinct representations where $d\ll O(k)$ seems to be a necessary next step for our approach.

As discussed in section 1, our techniques come from and generalize the catalytic computing framework despite being in a small space regime. Understanding the power of catalytic techniques to run many parallel d-ary products in the same space could help us understand the power of using catalytic techniques for small space classes, which could help us understand the power of small space.

ACKNOWLEDGMENTS

The authors would like to thank Stephen Cook and Toniann Pitassi for many helpful discussions leading up to this paper. We would also like to thank the anonymous reviewers for their useful suggestions. Both authors are partially funded by NSERC.

A IMPROVED d-ARY CATALYTIC PRODUCTS

As touched upon before, Lemma 7 gives a d-ary form of Lemma 5, with the main slowdown being the $O(2^d)$ recursive calls to each P_i program. As a greedy next step, we could try to reduce the number of recursive calls, possibly even to match the constant number needed in Lemma 5.

It is worth noting that we actually have such a construction which makes only $\operatorname{poly}(d)$ calls to each P_i, a major improvement on Lemma 7. We state and prove this improvement, and then discuss the problems with using the construction presented for improving our main results.

Lemma 10 (Lemma 7, polynomially efficient version). *Let $R_0,\ldots,R_d$ be distinct registers. Let $P_1\ldots P_d$ be a invertible programs where P_i updates*

$$R_i=\tau_i+v_i$$
$$R_j=\tau_j\quad\forall j\neq i.$$

Then there exists an invertible program P which updates

$$R_0=\tau_0+\prod_i v_i$$
$$R_j=\tau_j\quad\forall j\neq 0.$$

P uses the $d+1$ registers $R_0,\ldots,R_d$ plus d additional registers (not counting any space used by the programs P_i) and makes d^2 calls to each P_i plus $\operatorname{poly}(d)$ basic instructions of the form $R_p\leftarrow R_p\pm\prod_i R_i$.

Proof. We inductively compute $\prod_i v_i$ using Lemma 5 as a subroutine. We will compute the product like a binary tree, at each level j computing products of pairs from level $j-1$. Inductively each register at level j will be the product of 2^j f_i's. For all $j=0\ldots\log d$ let $\overrightarrow{R^j}=\{R_i^j\}$ be a vector of $\frac{d}{2^j}$ registers, where $R_i^0:=R_i$ for all

$i \in [d]$ and $R_1^{\log d} := R_0$. Since $\sum_j \frac{d}{2^j} = 2d$ this gives us $2d$ registers as claimed. Let $P_\ell^0 := P_1, P_3 \ldots$ and $P_r^0 := P_2, P_4 \ldots$, and for $j \in [\log d]$ we inductively define programs P_ℓ^j as follows:

```
1: $P_\ell^{j-1}$
2: for $i = 1, 3 \ldots$ do
3:     $R_i^j \leftarrow R_i^j - R_{2i-1}^{j-1} R_{2i}^{j-1}$
4: end for
5: $P_r^{j-1}$
6: for $i = 1, 3 \ldots$ do
7:     $R_i^j \leftarrow R_i^j + R_{2i-1}^{j-1} R_{2i}^{j-1}$
8: end for
9: $(P_\ell^{j-1})^{-1}$
10: for $i = 1, 3 \ldots$ do
11:     $R_i^j \leftarrow R_i^j - R_{2i-1}^{j-1} R_{2i}^{j-1}$
12: end for
13: $(P_r^{j-1})^{-1}$
14: for $i = 1, 3 \ldots$ do
15:     $R_i^j \leftarrow R_i^j + R_{2i-1}^{j-1} R_{2i}^{j-1}$
16: end for
```

We define the program P_r^j similarly but for even i in each loop instead.

We claim that P_ℓ^j (P_r^j) sets $R_i^j = \tau_i^j + \prod_{i'=2^j(i-1)+1}^{2^j i} v_{i'}$ for all odd (even) i and leaves all other registers untouched, and that it uses at most 4^j calls to each P_i plus $\frac{2}{3}(4^j - 1)d$ basic instructions. This is clear for $j = 0$ as P_ℓ^0 and P_r^0 simply add v_i to all the corresponding odd and even registers respectively using one call to each relevant P_i and no basic instructions. Inductively correctness follows by the correctness of the program for Lemma 5, as for each i our program performs the same steps. Since P_ℓ^{j-1} and P_r^{j-1} each make at most 4^{j-1} calls to each P_i and two calls are made to each of these programs, we get at most $4 \cdot 4^{j-1} = 4^j$ calls to each P_i program. The for loops add $2d$ basic instructions for a total of $4 \cdot \frac{2}{3}(4^{j-1} - 1)d + 2d = \frac{2}{3}(4^j - 1)d$.

Running $P_1^{\log d}$ we thus get $R_0 = R_1^{\log d} = \tau_0 + \prod_{i'=1}^{d} v_{i'}$ as required, with a total of $4^{\log d} = d^2$ calls to each P_i and $O(4^{\log d} d) = \text{poly}(d)$ basic instructions. □

This would seem like major leap for all our results: plugging these numbers into our construction for Theorem 2 would give a branching program with length $(\log k)^{3h}\,\text{poly}(\log k)$, which would go far and beyond the task of beating k^h for every super-constant k and h. Unfortunately, the recursive steps of Algorithms 2 and 3 (specifically, Lemmas 8 and 9) don't just compute one product $\prod_{i=1}^{d} v_i$ — they actually compute a sum involving k^2 different products. Recall that in Lemma 8 our key equation was

$$v_{p,b} = \sum_{(x,y,z) \in [k]\times[k]\times[k]} [x_b = 1][f_p(y,z) = x] \cdot \prod_{b' \in [\log k]} [v_{\ell,b'} = y_{b'}][v_{r,b'} = z_{b'}]$$

Each distinct (y, z) gives rise to a different product, because the y_b' or z_b' values will be distinct. This is no issue for Lemmas 8 and 9, because each different product in the sum is computed directly into the same R_v registers, whereas the tree-like construction in Lemma 10 uses $\log k$ extra registers, denoted R_i^j in the proof. Naïvely, then, computing all k^2 products simultaneously would require $k^2 \log k$ extra registers, at which point Algorithm 1 is a better option. It would be interesting to try to improve on this approach.

REFERENCES

[1] David A Barrington. 1989. Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1. *J. Comput. System Sci.* 38, 1 (1989), 150–164.

[2] Michael Ben-or and Richard Cleve. 1992. Computing Algebraic Formulas Using a Constant Number of Registers. *SIAM J. Comput.* 21, 1 (Feb. 1992), 54–58.

[3] Harry Buhrman, Richard Cleve, Michal Koucký, Bruno Loff, and Florian Speelman. 2014. Computing with a full memory: catalytic space. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*. ACM, 857–866.

[4] Harry Buhrman, Michal Koucký, Bruno Loff, and Florian Speelman. 2018. Catalytic Space: Non-determinism and Hierarchy. *Theory Comput. Syst.* 62, 1 (2018), 116–135.

[5] Diptarka Chakraborty, Debarati Das, Michal Koucký, and Nitin Saurabh. 2018. Space-Optimal Quasi-Gray Codes with Logarithmic Read Complexity. In *26th Annual European Symposium on Algorithms, ESA 2018, August 20-22, 2018, Helsinki, Finland (LIPIcs)*, Vol. 112. 12:1–12:15.

[6] Stephen Cook, Mark Braverman, Pierre McKenzie, Rahul Santhanam, and Dustin Wehr. 2009. Branching Programs: Avoiding Barriers. (August 2009). https://www.cs.toronto.edu/~sacook/barriers.ps Talk at Barriers Workshop at Princeton.

[7] Stephen Cook, Pierre McKenzie, Dustin Wehr, Mark Braverman, and Rahul Santhanam. 2012. Pebbles and Branching Programs for Tree Evaluation. *ACM Trans. Comput. Theory* 3, 2, Article 4 (Jan. 2012), 43 pages. https://doi.org/10.1145/2077336.2077337 arXiv version freely available at http://arxiv.org/abs/1005.2642.

[8] Samir Datta, Chetan Gupta, Rahul Jain, Vimal Raj Sharma, and Raghunath Tewari. 2020. Randomized and Symmetric Catalytic Computation. *Electronic Colloquium on Computational Complexity (ECCC)* 27 (2020), 24. https://eccc.weizmann.ac.il/report/2020/024

[9] Jeff Edmonds, Venkatesh Medabalimi, and Toniann Pitassi. 2018. Hardness of Function Composition for Semantic Read once Branching Programs. In *33rd Computational Complexity Conference, CCC 2018, June 22-24, 2018, San Diego, CA, USA (LIPIcs)*, Vol. 102. 15:1–15:22.

[10] Frank Gray. 1953. Pulse code communication. https://patents.google.com/patent/US2632058A/en. US Patent 2632058A.

[11] Chetan Gupta, Rahul Jain, Vimal Raj Sharma, and Raghunath Tewari. 2019. Unambiguous Catalytic Computation. *Electronic Colloquium on Computational Complexity (ECCC)* 26 (2019), 95.

[12] Michal Koucký. 2016. Catalytic computation. *Bulletin of the EATCS* 118 (2016).

[13] David Liu. 2013. Pebbling Arguments for Tree Evaluation. *CoRR* (2013).

[14] Michael S. Paterson and Carl E. Hewitt. 1970. Comparative Schematology. In *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*, Jack B. Dennis (Ed.). ACM, New York, NY, USA, 119–127. https://doi.org/10.1145/1344551.1344563

[15] Aaron Potechin. 2016. A Note on Amortized Branching Program Complexity. arXiv:cs.CC/1611.06632

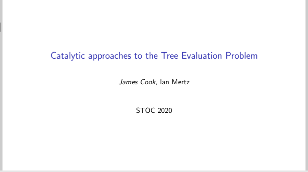
[16] John H. Reif and Stephen R. Tate. 1992. ON THRESHOLD CIRCUITS AND POLYNOMIAL COMPUTATION.

Catalytic approaches to the Tree Evaluation Problem

James Cook, Ian Mertz

STOC 2020

Catalytic approaches to the Tree Evaluation Problem

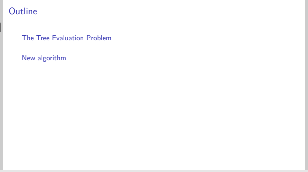

Catalytic approaches to the Tree Evaluation Problem

James Cook, Ian Mertz

STOC 2020

Hello from Toronto, Canada!
This video is an overview of a paper by Ian Mertz and myself, about a new space-efficient algorithm for the Tree Evaluation Problem.
The video is divided into two parts.

Outline

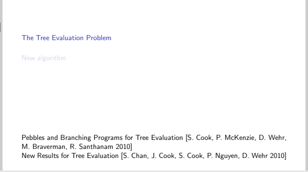

Outline

In the first part, I'll talk about the Tree Evaluation problem. It was introduced in an attempt to separate complexity classes: the problem can easily be solved in polynomial time, but it seems impossible to solve in low-memory classes like log space.
In the second part, I'll show you a new algorithm for solving this problem with limited memory. This algorithm gives the first space improvement since the problem was originally introduced ten years ago, and it makes use of some techniques for re-using memory using reversible computations.

The Tree Evaluation Problem

New algorithm

Pebbles and Branching Programs for Tree Evaluation [S. Cook, P. McKenzie, D. Wehr, M. Braverman, R. Santhanam 2010]
New Results for Tree Evaluation [S. Chan, J. Cook, S. Cook, P. Nguyen, D. Wehr 2010]

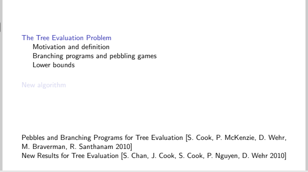

The Tree Evaluation Problem

New algorithm

Pebbles and Branching Programs for Tree Evaluation [S. Cook, P. McKenzie, D. Wehr, M. Braverman, R. Santhanam 2010]
New Results for Tree Evaluation [S. Chan, J. Cook, S. Cook, P. Nguyen, D. Wehr 2010]

The first part is older work mostly done by other people.
It’s based on a couple of papers from 2010 that introduced the problem.

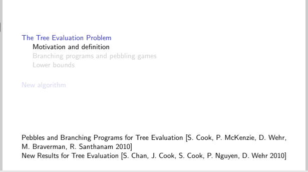
Pebbles and Branching Programs for Tree Evaluation [S. Cook, P. McKenzie, D. Wehr, M. Braverman, R. Santhanam 2010]
New Results for Tree Evaluation [S. Chan, J. Cook, S. Cook, P. Nguyen, D. Wehr 2010]



Pebbles and Branching Programs for Tree Evaluation [S. Cook, P. McKenzie, D. Wehr, M. Braverman, R. Santhanam 2010]
New Results for Tree Evaluation [S. Chan, J. Cook, S. Cook, P. Nguyen, D. Wehr 2010]

I'll start by describing the problem and its motivation. Then I'll talk about a couple of abstractions we use to analyse it, called branching programs and pebbling games. And finally, before I move on the new algorithm, I'll talk about some lower bounds that the our algorithm had to work around.

Pebbles and Branching Programs for Tree Evaluation [S. Cook, P. McKenzie, D. Wehr, M. Braverman, R. Santhanam 2010]
New Results for Tree Evaluation [S. Chan, J. Cook, S. Cook, P. Nguyen, D. Wehr 2010]



Pebbles and Branching Programs for Tree Evaluation [S. Cook, P. McKenzie, D. Wehr, M. Braverman, R. Santhanam 2010]
New Results for Tree Evaluation [S. Chan, J. Cook, S. Cook, P. Nguyen, D. Wehr 2010]

So, let’s start with the motivation.

The Tree Evaluation Problem (TEP)

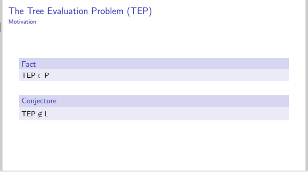
Motivation

Fact

$\text{TEP} \in \text{P}$

Conjecture

$\text{TEP} \notin \text{L}$

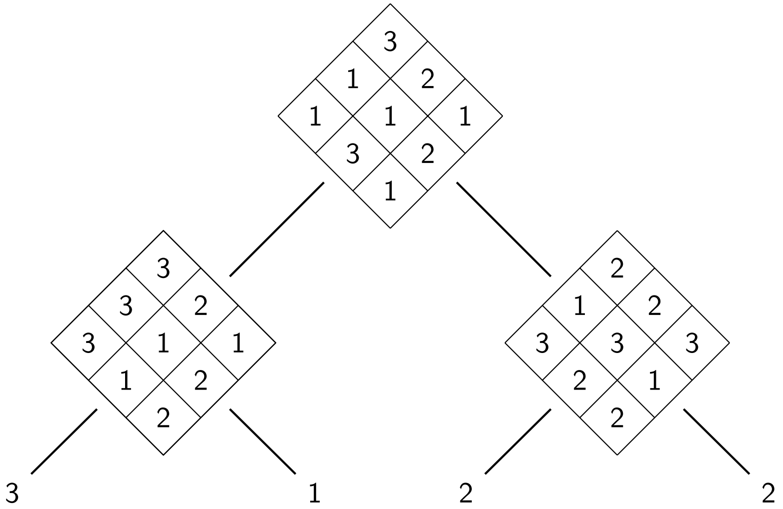

The Tree Evaluation Problem (TEP)
Motivation

Fact
TEP $\in$ P

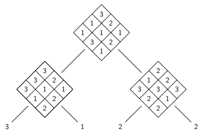
Conjecture
TEP $\notin$ L

The Tree Evaluation Problem, or TEP for short is easy to solve in polynomial time, but it's conjectured that you can't solve it in log space. The goal is to prove this conjecture, implying that L is not equal to P. In fact, the gap it aims to close is a little narrower than that: it's in log CFL but conjectured not to be in NL. But we'll just focus on P and L in this video.
So, that's the motivation. Now let's talk about what this problem actually is.

The Tree Evaluation Problem (TEP)

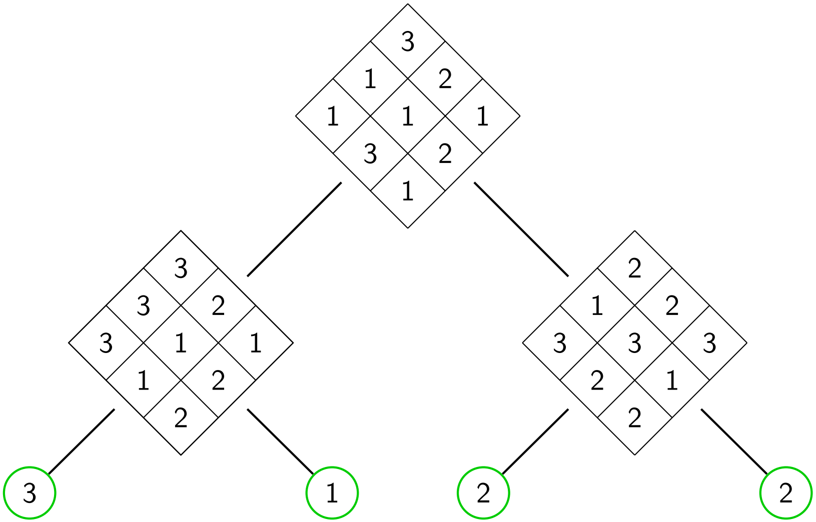


The Tree Evaluation Problem (TEP)

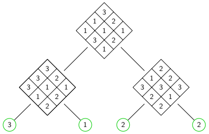


The input to the Tree Evaluation Problem is a complete binary tree with some information attached to each node. Each leaf has a number attached to it — in this case, 3, 1, 2 and 2 — and each internal node has a table of numbers.
Given that input, we're going to recursively define a single number at each node, called the value of the node.

The Tree Evaluation Problem (TEP)

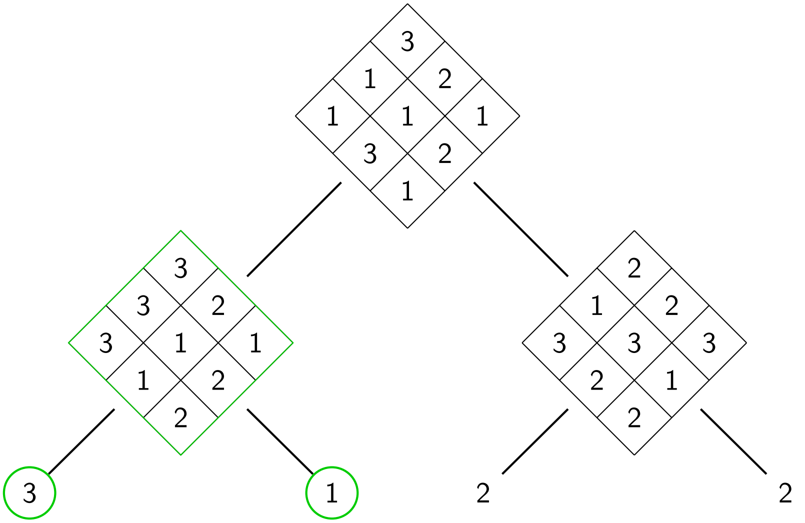


The Tree Evaluation Problem (TEP)

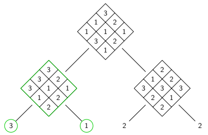


The values of the leaves are already part of the input.
To compute the value of an internal node, we need to first know the values of its children.

The Tree Evaluation Problem (TEP)

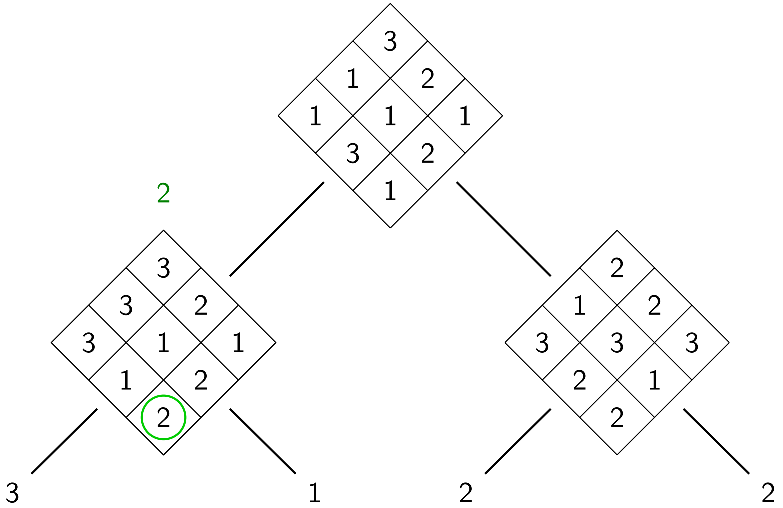


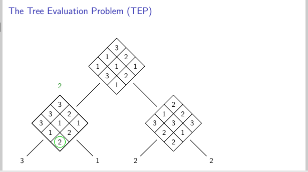
The Tree Evaluation Problem (TEP)



For example, let's look at the left child of the root. The values of its two children tell us where to look in its table. In this case, we look at row three, column one, and we find the number two.

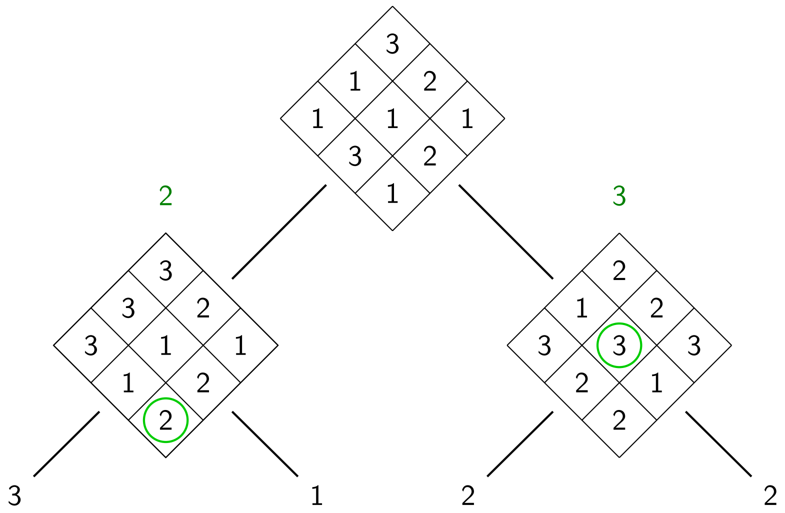
The Tree Evaluation Problem (TEP)

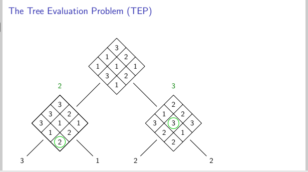



The Tree Evaluation Problem (TEP)

Similarly, we look up row two column two of the node on the right, and find the number three.

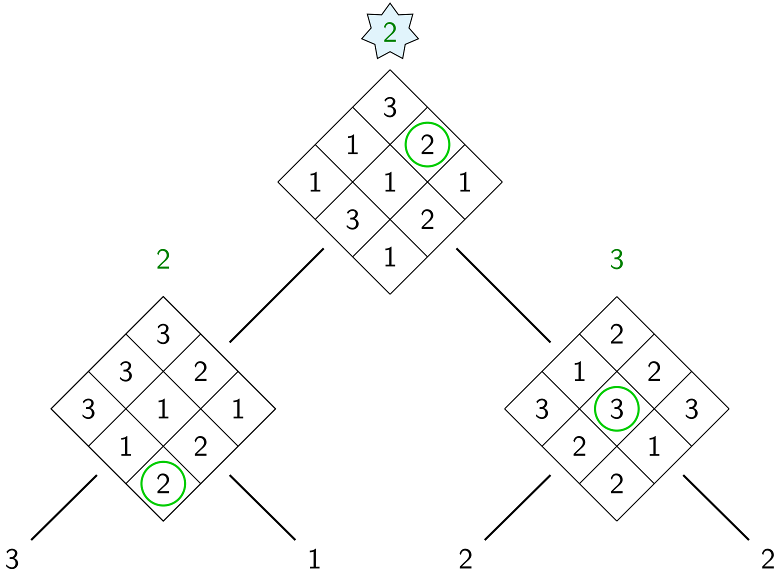
The Tree Evaluation Problem (TEP)



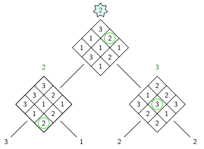


Finally, the numbers two and three tell us where to look in the root node, and we find the number two.

The Tree Evaluation Problem (TEP)

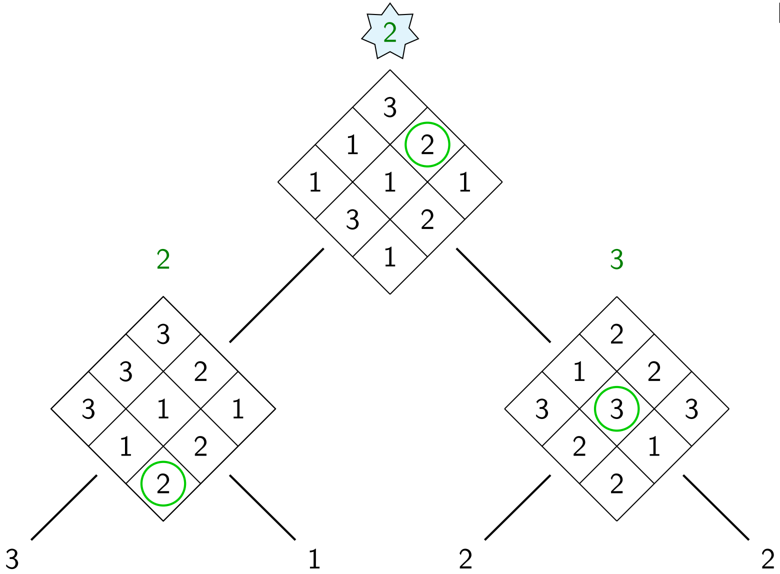


The Tree Evaluation Problem (TEP)



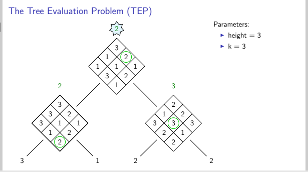
The output of the Tree Evaluation Problem is the value at the root.

The Tree Evaluation Problem (TEP)



Parameters:

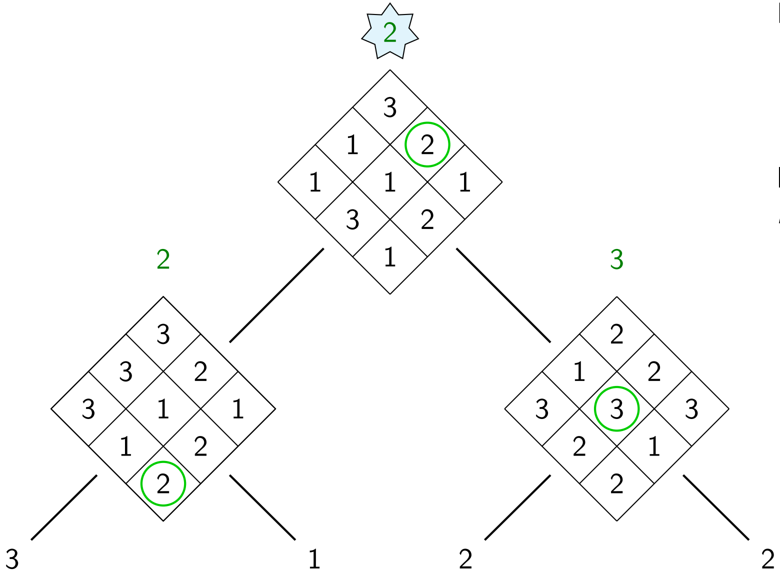
- height = 3
- k = 3


The Tree Evaluation Problem (TEP)

Parameters:
- height = 3
- k = 3

There are two parameters to this problem. The first is the height of the tree. Three in this case. The second parameter is k, which is the range of the numbers at the nodes. In this case it's also three, meaning every number is between one and three, and the tables are all three by three.

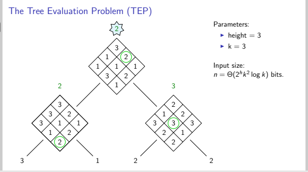
The Tree Evaluation Problem (TEP)



Parameters:

- height $= 3$
- k $= 3$

Input size:
$n = \Theta(2^h k^2 \log k)$ bits.



The Tree Evaluation Problem (TEP)

Parameters:

- height = 3
- k = 3

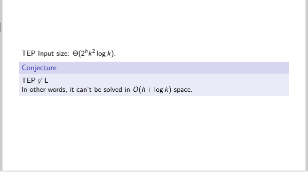
Input size:
$n = \Theta(2^h k^2 \log k)$ bits.

The size of the input to TEP is on the order of two to the h internal nodes, times k squared numbers stored in each node, times log k bits to store each number.

TEP Input size: $\Theta(2^h k^2 \log k)$.

Conjecture

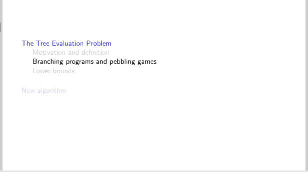
TEP $\notin$ L
In other words, it can't be solved in $O(h + \log k)$ space.


TEP Input size: $\Theta(2^h k^2 \log k)$.

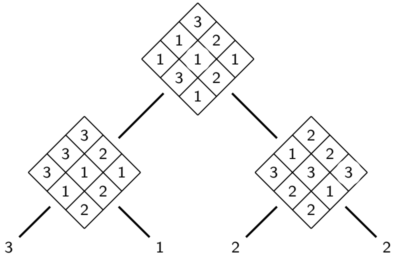
Conjecture

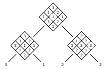
TEP $\notin$ L
In other words, it can't be solved in $O(h + \log k)$ space.

Using this formula for the input size, we can rephrase the conjecture I showed you earlier.
Saying TEP is not in L is the same as saying it can't be solved in big oh of h plus log k space.

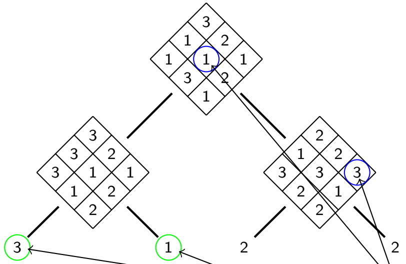


Now that I've defined the Tree Evaluation Problem, I want to talk about algorithms for solving it. I'll start by describing branching programs, which are the computational model we're using. Then I'll talk about an abstraction called a *pebbling game* which can be useful for both upper and lower bounds.

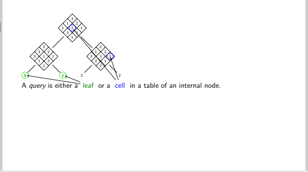

3
1
2
1
1
1
3
2
1
3
3
2
3
1
1
1
2
2
2
1
2
3
3
3
2
1
2
3
1
2
2



So, here's our TEP input again. I'll define a *query* to be any piece of that input we might want to read.

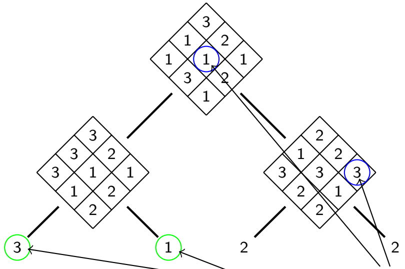


A *query* is either a leaf or a cell in a table of an internal node.



A *query* is either a leaf or a cell in a table of an internal node.

Specifically, a query is either a leaf, meaning we want to read the input at that leaf, or it’s a particular cell in one of the tables in an internal node.

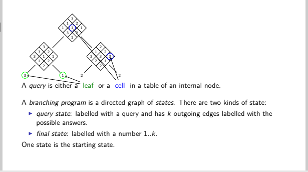


A *query* is either a leaf or a cell in a table of an internal node.

A *branching program* is a directed graph of *states*. There are two kinds of state:

- *query state*: labelled with a query and has k outgoing edges labelled with the possible answers.
- *final state*: labelled with a number $1..k$.

One state is the starting state.



A *query* is either a leaf or a cell in a table of an internal node.

A *branching program* is a directed graph of *states*. There are two kinds of state:

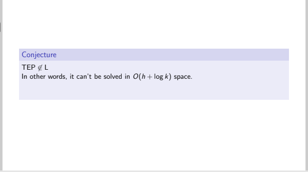
- *query state*: labelled with a query and has k outgoing edges labelled with the possible answers.
- *final state*: labelled with a number $1..k$.

One state is the starting state.

A branching program is a directed graph, where the nodes are called states. There are two kinds of state.
A query state is labelled with a query, and has k outgoing edges: the edge you follow depends on the answer to the query.
The other kind is a final state. When you get to one of those, the computation stops, and you output whatever the state is labelled with.
One of the states is marked as the starting state, where computation begins.

Conjecture

TEP $\notin$ L
In other words, it can't be solved in $O(h + \log k)$ space.



Conjecture

TEP $\notin$ L
In other words, it can't be solved in $O(h + \log k)$ space.

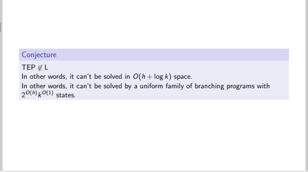
Let's return to our lower bound conjecture. We've written it as: TEP can't be solved in big oh of h plus log k space.
Any Turing machine can be transformed into a uniform family of branching programs, with one state for each possible configuration.

Conjecture

$\mathrm{TEP} \notin \mathrm{L}$

In other words, it can't be solved in $O(h + \log k)$ space.

In other words, it can't be solved by a uniform family of branching programs with $2^{O(h)}k^{O(1)}$ states.



Conjecture

TEP $\notin$ L

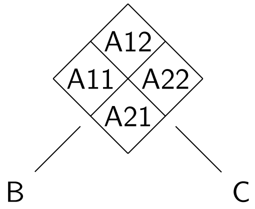
In other words, it can't be solved in $O(h + \log k)$ space.

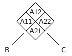
In other words, it can't be solved by a uniform family of branching programs with $2^{O(h)}k^{O(1)}$ states.

So, we can rephrase our conjecture one more time: TEP can't be solved by a uniform family of branching programs with only two to the order h times a polynomial in k states. We could also state the conjecture without the uniformity condition.

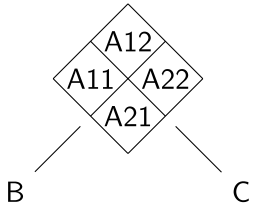
Now, let's look at an example of a branching program for solving TEP.

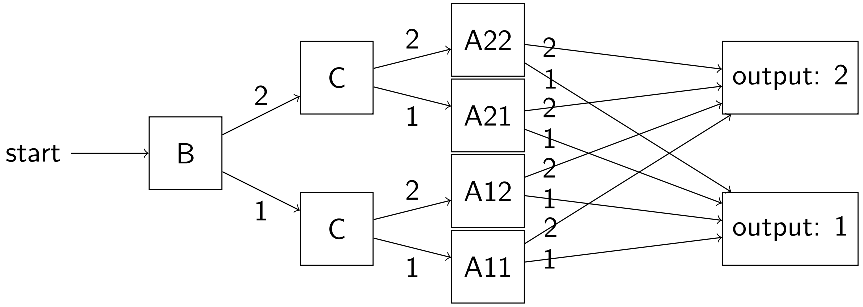
To keep it small, we'll set both the height and the alphabet size to 2.

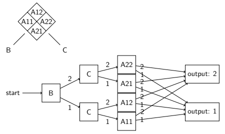

A12
A11
A22
A21
B
C



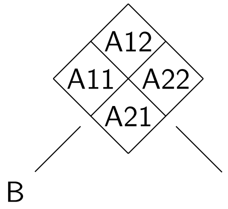
When both h and k are two, an input to TEP is structured like this. There are six things we can query: the four cells in the root node A's table, and the two leaves B and C.

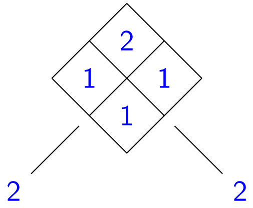

A12
A11
A22
A21
B
C

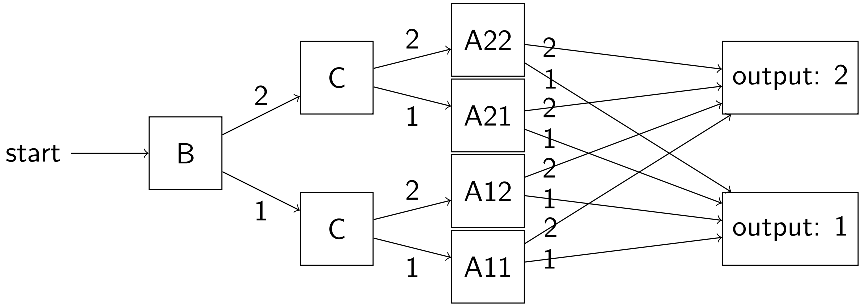

start
B
2
1
C
C
2
1
2
1
A22
A21
A12
A11
2
1
2
1
2
1
2
1
output: 2
output: 1

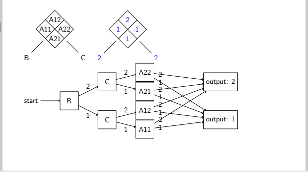


Here's a branching program that solves it. It's organized into *layers* going from left to right.
The starting state queries the first leaf, B. Depending on the answer, we end up in one of the two states in the next layer. Those states query the other leaf C, and depending on the answer, we end up in one of four possible states in the third layer. Each node in the third layer queries a different cell in the root node's table, and depending on the answer, we output 1 or 2.

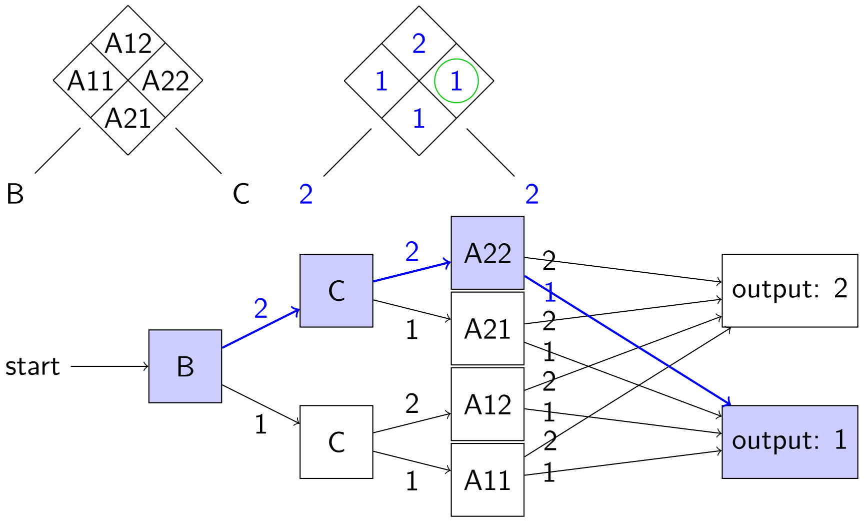

A12
A11
A22
A21
B
C

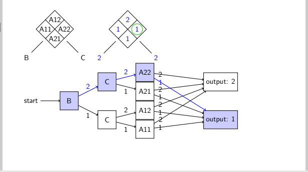

2
1
1
1
2
2


start
B
C
C
A22
A21
A12
A11
output: 2
output: 1
2
1
2
1
2
1
2
1
2
1
2
1
2
1

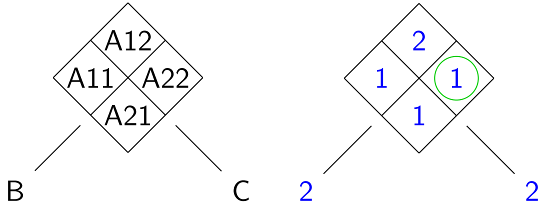


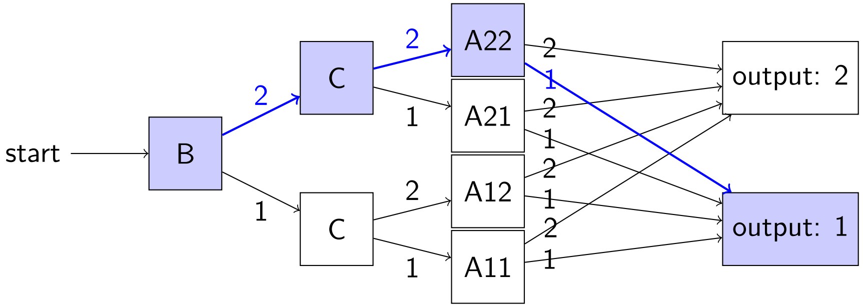
Here's an example input. Let's see what the computation looks like.


A12
A11
A22
A21
B
C
2
1
1
1
2
2
start
B
2
1
C
C
2
1
2
1
A22
A21
A12
A11
2
1
2
1
2
1
2
1
output: 2
output: 1



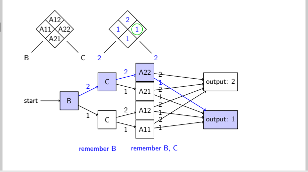
Both the leaves are 2, so we end up at the node that queries A22. Then the value is 1, so we output 1.
One thing to notice here is that every layer remembers a different set of information.





remember B

remember B, C



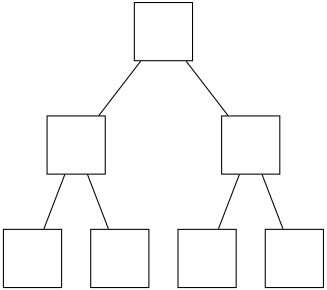
In the second layer, we remember node B, and in the third layer, we remember both B and C. All the lower bounds we have so far for TEP involve arguments about how many things the branching program needs to remember at once.
One way to model this idea of remembering things is pebbling games.

Pebbling game [Paterson Hewitt 1970]

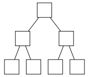
Pebbling game [Paterson Hewitt 1970]

Pebbling games were first defined by Paterson and Hewitt in 1970. In the context of the Tree Evaluation Problem, they work like this. Suppose we have a complete binary tree of height h.

Pebbling game [Paterson Hewitt 1970]

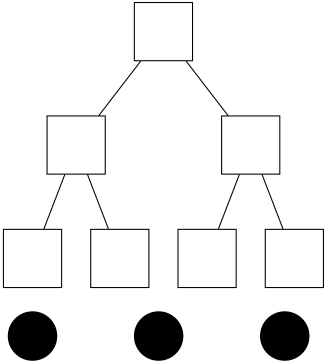


Pebbling game [Paterson Hewitt 1970]

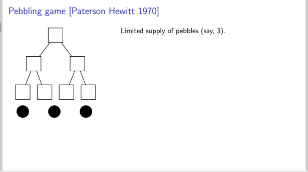


Three in this case.

Pebbling game [Paterson Hewitt 1970]



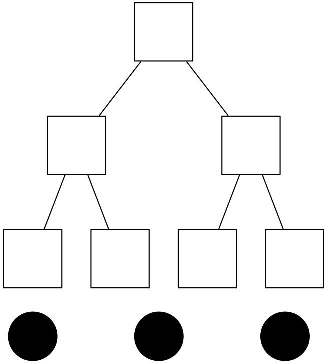
Limited supply of pebbles (say, 3).


Pebbling game [Paterson Hewitt 1970]

Limited supply of pebbles (say, 3).

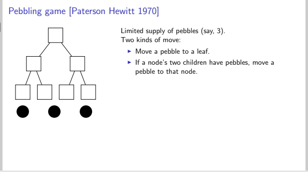
You have some limited number of pebbles. Let's say it's also three. They all start in your hand. You're allowed two kinds of move.

Pebbling game [Paterson Hewitt 1970]

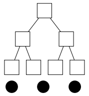


Limited supply of pebbles (say, 3).
Two kinds of move:

- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.



Pebbling game [Paterson Hewitt 1970]

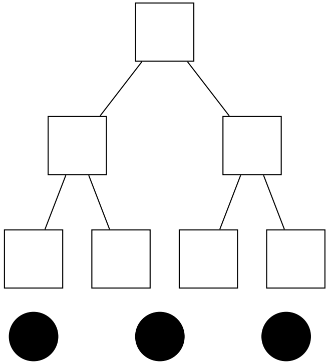


Limited supply of pebbles (say, 3).
Two kinds of move:

- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

First, you can move one of your pebbles to a leaf of the tree. And second, if a node's two children both have pebbles on them, you can move one of your pebbles to that node. The goal is to place a pebble on the root node.

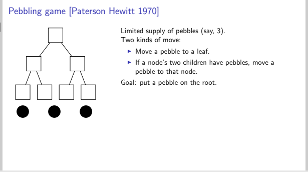
Pebbling game [Paterson Hewitt 1970]



Limited supply of pebbles (say, 3).
Two kinds of move:

- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

Goal: put a pebble on the root.


Pebbling game [Paterson Hewitt 1970]

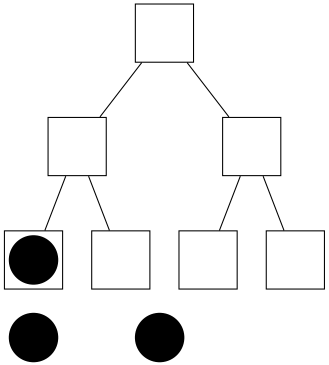
Limited supply of pebbles (say, 3).
Two kinds of move:

- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

Goal: put a pebble on the root.

Here's a sequence of moves that does this. We start with the leaves and work our way up.

Pebbling game [Paterson Hewitt 1970]

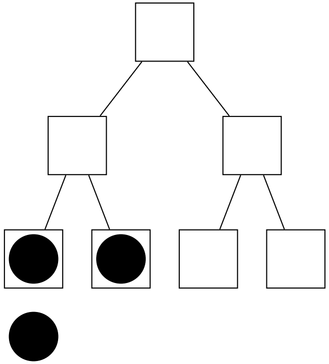


Limited supply of pebbles (say, 3).
Two kinds of move:

- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

Goal: put a pebble on the root.

Pebbling game [Paterson Hewitt 1970]

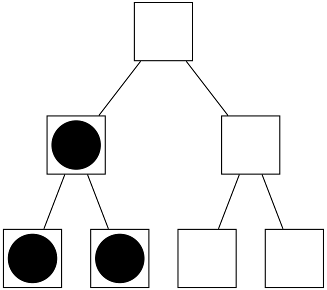


Limited supply of pebbles (say, 3).
Two kinds of move:

- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

Goal: put a pebble on the root.

Pebbling game [Paterson Hewitt 1970]

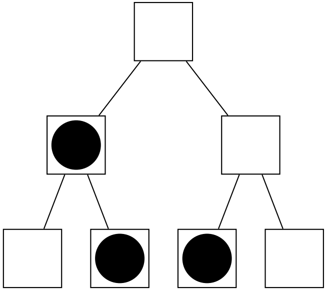


Limited supply of pebbles (say, 3).
Two kinds of move:

- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

Goal: put a pebble on the root.

Pebbling game [Paterson Hewitt 1970]

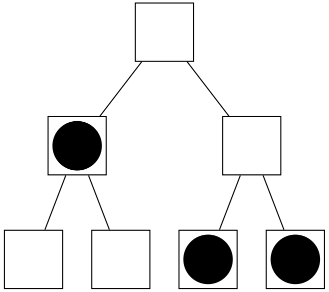


Limited supply of pebbles (say, 3).
Two kinds of move:

- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

Goal: put a pebble on the root.

Pebbling game [Paterson Hewitt 1970]

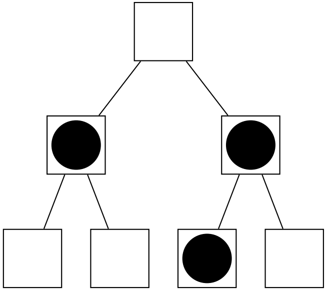


Limited supply of pebbles (say, 3).
Two kinds of move:

- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

Goal: put a pebble on the root.

Pebbling game [Paterson Hewitt 1970]



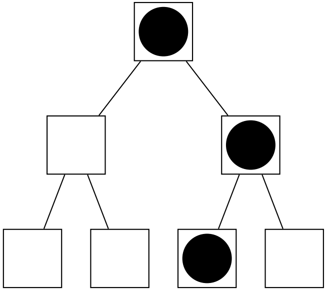
Limited supply of pebbles (say, 3).

Two kinds of move:

- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

Goal: put a pebble on the root.

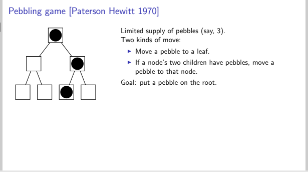
Pebbling game [Paterson Hewitt 1970]



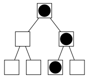
Limited supply of pebbles (say, 3).
Two kinds of move:

- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

Goal: put a pebble on the root.



Pebbling game [Paterson Hewitt 1970]



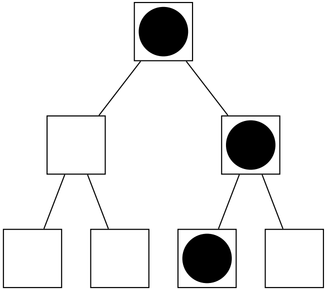
Limited supply of pebbles (say, 3).
Two kinds of move:

- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

Goal: put a pebble on the root.

We've succeeded, because there's now a pebble on the root node. The important question is: how many pebbles do we need? In this case we had three pebbles, and it was enough.

Pebbling game [Paterson Hewitt 1970]

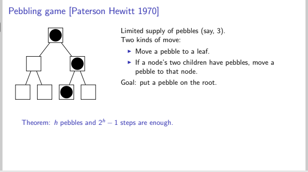


Limited supply of pebbles (say, 3).
Two kinds of move:

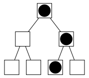
- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

Goal: put a pebble on the root.

Theorem: h pebbles and $2^h - 1$ steps are enough.



Pebbling game [Paterson Hewitt 1970]



Limited supply of pebbles (say, 3).
Two kinds of move:

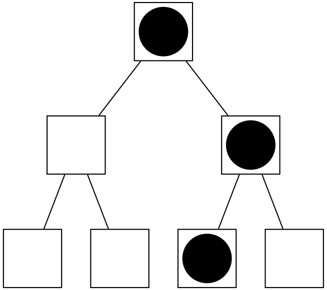
- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

Goal: put a pebble on the root.

Theorem: h pebbles and $2^h - 1$ steps are enough.

In general, you can solve this game with h pebbles, where h is the height of the tree, using a simple recursive algorithm. The algorithm visits each node once, so that's two to the h minus one steps.

Pebbling game [Paterson Hewitt 1970]

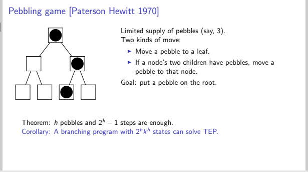


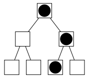
Limited supply of pebbles (say, 3).
Two kinds of move:

- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

Goal: put a pebble on the root.

Theorem: h pebbles and $2^h - 1$ steps are enough.
Corollary: A branching program with $2^h k^h$ states can solve TEP.


Pebbling game [Paterson Hewitt 1970]



Limited supply of pebbles (say, 3).
Two kinds of move:

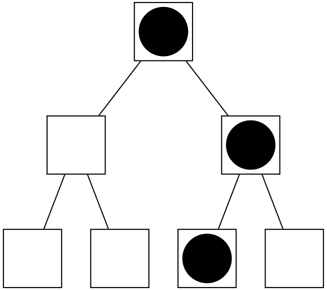
- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

Goal: put a pebble on the root.

Theorem: h pebbles and $2^h - 1$ steps are enough.
Corollary: A branching program with $2^h k^h$ states can solve TEP.

A corollary of that is that we can build a branching program that solves the tree evaluation problem using two to the h times k to the h states. Each step of the game translates into a layer of the branching program, and the placement of the pebbles determines which values the program is remembering. Since our strategy uses at most h pebbles at a time, the program will only need to remember at most h values at once, which requires k to the power h states in a single layer.
Now, the pebbling strategy is tight: if you only have h-1 pebbles, no sequence of legal moves can put one on the root.

Pebbling game [Paterson Hewitt 1970]



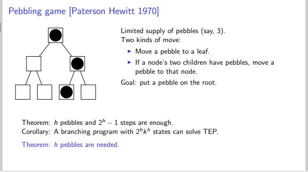
Limited supply of pebbles (say, 3).
Two kinds of move:

- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

Goal: put a pebble on the root.

Theorem: h pebbles and $2^h - 1$ steps are enough.
Corollary: A branching program with $2^h k^h$ states can solve TEP.

Theorem: h pebbles are needed.



Pebbling game [Paterson Hewitt 1970]

Limited supply of pebbles (say, 3).
Two kinds of move:

- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

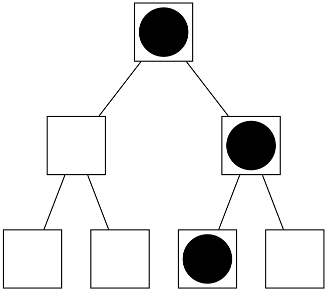
Goal: put a pebble on the root.

Theorem: h pebbles and $2^h - 1$ steps are enough.
Corollary: A branching program with $2^h k^h$ states can solve TEP.

Theorem: h pebbles are needed.

The proof of that is not as obvious. I'll leave it as an exercise. Now, it would be nice if we could make a corresponding corollary.

Pebbling game [Paterson Hewitt 1970]



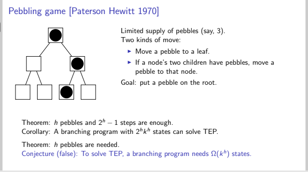
Limited supply of pebbles (say, 3).
Two kinds of move:

- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

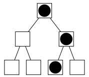
Goal: put a pebble on the root.

Theorem: h pebbles and $2^h - 1$ steps are enough.
Corollary: A branching program with $2^h k^h$ states can solve TEP.

Theorem: h pebbles are needed.
Conjecture (false): To solve TEP, a branching program needs $\Omega(k^h)$ states.



Pebbling game [Paterson Hewitt 1970]



Limited supply of pebbles (say, 3).
Two kinds of move:

- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

Goal: put a pebble on the root.

Theorem: h pebbles and $2^h - 1$ steps are enough.
Corollary: A branching program with $2^h k^h$ states can solve TEP.

Theorem: h pebbles are needed.
Conjecture (false): To solve TEP, a branching program needs $\Omega(k^h)$ states.

Since we need at least h pebbles, maybe we can prove that the tree evaluation problem needs at least on the order of k to the h states. We'll see in a moment that this would imply that log space is not equal to polytime.
For a long time, nobody could come up with any algorithm that did better, so this conjecture seemed quite plausible.
The algorithm I'll present later is the first counterexample.
Let's take a look at where we are.

Conjecture (TEP $\notin$ L)

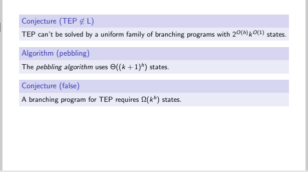
TEP can't be solved by a uniform family of branching programs with $2^{O(h)}k^{O(1)}$ states.

Algorithm (pebbling)

The *pebbling algorithm* uses $\Theta((k+1)^h)$ states.

Conjecture (false)

A branching program for TEP requires $\Omega(k^h)$ states.



Conjecture (TEP $\notin$ L)

TEP can't be solved by a uniform family of branching programs with $2^{O(h)} k^{O(1)}$ states.

Algorithm (pebbling)

The *pebbling algorithm* uses $\Theta((k+1)^h)$ states.

Conjecture (false)

A branching program for TEP requires $\Omega(k^h)$ states.

We started with this conjecture that TEP is not in L, meaning two to the order h times poly k states isn't enough.
We saw our first algorithm. If you analyse it carefully, it turns out the pebbling algorithm uses on the order of k plus one to the power h states.
And the pebbling framework led to a conjuctered lower bound of k to the h.

Conjecture (TEP $\notin$ L)

TEP can't be solved by a uniform family of branching programs with $2^{O(h)}k^{O(1)}$ states.

Algorithm (pebbling)

The *pebbling algorithm* uses $\Theta((k+1)^h)$ states.

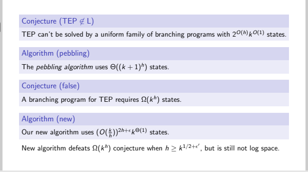
Conjecture (false)

A branching program for TEP requires $\Omega(k^h)$ states.

Algorithm (new)

Our new algorithm uses $(O(\frac{k}{h}))^{2h+\epsilon}k^{\Theta(1)}$ states.

New algorithm defeats $\Omega(k^h)$ conjecture when $h \geq k^{1/2+\epsilon'}$, but is still not log space.


Conjecture (TEP $\notin$ L)

TEP can't be solved by a uniform family of branching programs with $2^{O(h)}k^{O(1)}$ states.

Algorithm (pebbling)

The *pebbling algorithm* uses $\Theta((k+1)^h)$ states.

Conjecture (false)

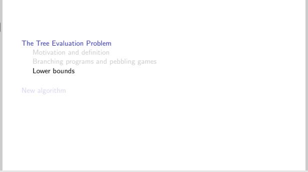
A branching program for TEP requires $\Omega(k^h)$ states.

Algorithm (new)

Our new algorithm uses $(O(\frac{k}{h}))^{2h+\epsilon}k^{\Theta(1)}$ states.

New algorithm defeats $\Omega(k^h)$ conjecture when $h \geq k^{1/2+\epsilon'}$, but is still not log space.

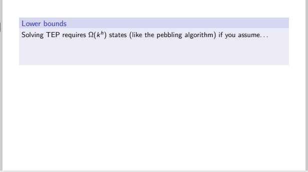
The new algorithm I'm going to show you has on the order of k over h to the power two h plus an arbitrarily small constant times a polynomial in k states.
This defeats the conjectured lower bound of k to the h whenever h is not too small compared to k. Specifically, if h is k to a power bigger than one half, this algorithm is an asymptotic improvement.
But, it's still not a log space algorithm, so the door is still open to using TEP as a way to separate L from P.



Now, I want to briefly mention some existing lower bounds for TEP, to give you an idea of why we found this new algorithm surprising.

Lower bounds

Solving TEP requires $\Omega(k^h)$ states (like the pebbling algorithm) if you assume. . .

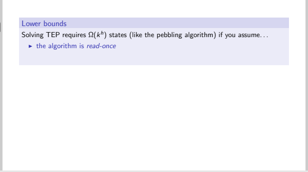

Lower bounds
Solving TEP requires $\Omega(k^h)$ states (like the pebbling algorithm) if you assume...

It turns out that under some pretty reasonable-sounding assumptions, you can prove that the pebbling algorithm is essentially the best possible.

Lower bounds

Solving TEP requires $\Omega(k^h)$ states (like the pebbling algorithm) if you assume. . .

- the algorithm is *read-once*


Lower bounds

Solving TEP requires $\Omega(k^h)$ states (like the pebbling algorithm) if you assume...

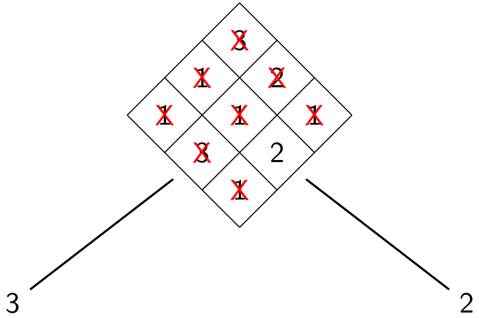
- the algorithm is *read-once*

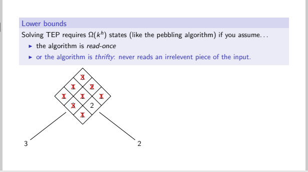
You can prove it if you assume the algorithm is *read-once*. That means that once the algorithm reads a certain piece of the input, it is not allowed to read it again.

Lower bounds

Solving TEP requires $\Omega(k^h)$ states (like the pebbling algorithm) if you assume...

- the algorithm is *read-once*
- or the algorithm is *thrifty*: never reads an irrelevent piece of the input.





Lower bounds

Solving TEP requires $\Omega(k^h)$ states (like the pebbling algorithm) if you assume...

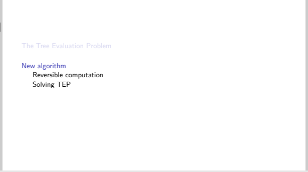
- the algorithm is *read-once*
- or the algorithm is *thrifty*: never reads an irrelevent piece of the input.

Another assumption we can make instead is that the algorithm is *thrifty*. This means that the algorithm never reads an irrelevant piece of the input. For example, if an internal node's left child has value three and its right child has value 2, then it's only allowed to read the entry at position three two in that node's table, since none of the other entries matter. Our new algorithm beats this lower bound of k to the h, so, as you may have already inferred, it's not read-once or thrifty. Our algorithm is actually going to read every piece of the input several times. I've said a lot of mysterious things about the algorithm, so maybe it's time I told you how it works.

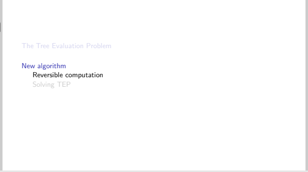
The Tree Evaluation Problem

New algorithm

- Reversible computation
- Solving TEP



This part of the video has two pieces.
I'll begin with some techniques we use related to reversible computation,
and then I'll tell you how we apply them to solve TEP.



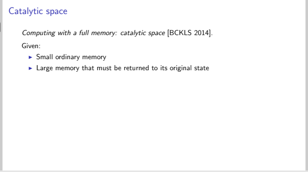
The first thing I want to tell you about is a paper that caught our attention, and showed us that reversible computation is something we should be looking at.

Catalytic space

Computing with a full memory: catalytic space [BCKLS 2014].

Given:

- Small ordinary memory
- Large memory that must be returned to its original state



Catalytic space

Computing with a full memory: catalytic space [BCKLS 2014].

Given:

- Small ordinary memory
- Large memory that must be returned to its original state

The paper is from 2014, and it's called *Computing with a full memory: catalytic space*.
The idea is that you're given a small amount of ordinary memory to work with, and a much larger amount of extra memory. The catch with the extra memory is that it starts out filled with data, possibly incompressible, and once you're done with your computation, you need to return it back the way it was.
Surprisingly, the authors found that the extra memory seems to help.

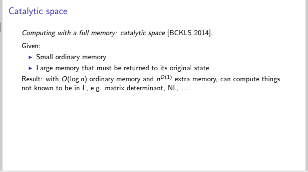
Catalytic space

Computing with a full memory: catalytic space [BCKLS 2014].

Given:

- Small ordinary memory
- Large memory that must be returned to its original state

Result: with $O(\log n)$ ordinary memory and $n^{O(1)}$ extra memory, can compute things not known to be in L, e.g. matrix determinant, NL, ...



Catalytic space

Computing with a full memory: catalytic space [BCKLS 2014].

Given:

- Small ordinary memory
- Large memory that must be returned to its original state

Result: with $O(\log n)$ ordinary memory and $n^{O(1)}$ extra memory, can compute things not known to be in L, e.g. matrix determinant, NL, ...

With only a logarithmic amount of ordinary memory but a polynomial amount of borrowed memory, you can compute many things not known to be computable in log space, such as the determinant of a matrix, or anything in nondeterministic log space.
We stumbled on this result when we were trying to prove a lower bound for the Tree Evaluation Problem.

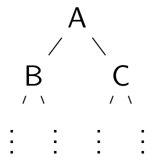
Catalytic space

Computing with a full memory: catalytic space [BCKLS 2014].

Given:

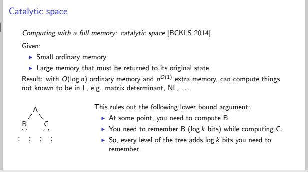
- Small ordinary memory
- Large memory that must be returned to its original state

Result: with $O(\log n)$ ordinary memory and $n^{O(1)}$ extra memory, can compute things not known to be in L, e.g. matrix determinant, NL, . . .



This rules out the following lower bound argument:

- At some point, you need to compute B.
- You need to remember B ($\log k$ bits) while computing C.
- So, every level of the tree adds $\log k$ bits you need to remember.



Catalytic space

Computing with a full memory: catalytic space [BCKLS 2014].

Given:

- Small ordinary memory
- Large memory that must be returned to its original state

Result: with $O(\log n)$ ordinary memory and $n^{O(1)}$ extra memory, can compute things not known to be in L, e.g. matrix determinant, NL, . . .

This rules out the following lower bound argument:

- At some point, you need to compute B.
- You need to remember B ($\log k$ bits) while computing C.
- So, every level of the tree adds $\log k$ bits you need to remember.

We had the following idea for a proof. First, at some point you need to compute the left child of the root: node B in this diagram. Then you need to keep that in memory while you compute the right child, C. That uses up log k bits of memory in addition to the subroutine that's computing C. Therefore, the argument goes, every level you add to the tree adds log k bits that your algorithm needs to remember.

The catalytic space result effectively shows that this approach will never work. Even if we could argue that you need to remember B while you're computing C, this result says that the subroutine computing C can borrow the memory being used to store B.

The history of the techniques we use goes back pretty far.

Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1*.* [D. Barrington 1989]

Computing algebraic formulas using a constant number of registers. [M. Ben-Or, R. Cleve 1992]

Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1. [D. Barrington 1989]

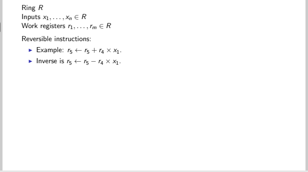
Computing algebraic formulas using a constant number of registers. [M. Ben-Or, R. Cleve 1992]

A 1989 paper by Barrington showed that if you restrict branching programs to have just five nodes in every layer, you can still do a lot with them. A later 1992 paper by Ben-Or and Cleve showed how you can do a lot with register programs that only use three registers.
Both of these papers show how you can trade time for space in order to make algorithms that use an extremely limited amount of memory.
Another thing they have in common is that they use reversible operations.
The basic ingredient in our algorithm is reversible operations on registers.

Ring R
Inputs $x_1, \ldots, x_n \in R$
Work registers $r_1, \ldots, r_m \in R$

Reversible instructions:

- Example: $r_5 \leftarrow r_5 + r_4 \times x_1$.
- Inverse is $r_5 \leftarrow r_5 - r_4 \times x_1$.



Ring R
Inputs $x_1, \dots, x_n \in R$
Work registers $r_1, \dots, r_m \in R$

Reversible instructions:

- Example: $r_5 \leftarrow r_5 + r_4 \times x_1$.
- Inverse is $r_5 \leftarrow r_5 - r_4 \times x_1$.

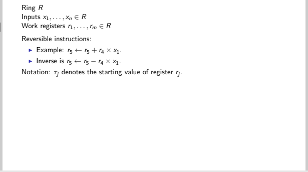
The model is that we have n inputs, x one through x n, and m work registers r one through r m, and their values are all in some ring R. We're interested in reversible instructions. For example, the first instruction here adds register four times input 1 to register five. We can reverse that instruction by subtracting instead of adding. When you run these two instructions in sequence, it's the same as doing nothing.

Ring R
Inputs $x_1, \ldots, x_n \in R$
Work registers $r_1, \ldots, r_m \in R$

Reversible instructions:

- Example: $r_5 \leftarrow r_5 + r_4 \times x_1$.
- Inverse is $r_5 \leftarrow r_5 - r_4 \times x_1$.

Notation: τ_j denotes the starting value of register r_j.


Ring R
Inputs $x_1, \dots, x_n \in R$
Work registers $r_1, \dots, r_m \in R$

Reversible instructions:

- Example: $r_5 \leftarrow r_5 + r_4 \times x_1$.
- Inverse is $r_5 \leftarrow r_5 - r_4 \times x_1$.

Notation: τ_j denotes the starting value of register r_j.

For any register r_j, let's define τ_j to be its initial value before our computation begins.
Now, suppose we have some function f we're interested in computing.
I'm going to define something called *cleanly computing* f.

Ring R
Inputs $x_1, \ldots, x_n \in R$
Work registers $r_1, \ldots, r_m \in R$

Reversible instructions:

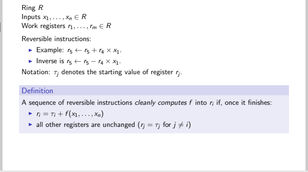
- Example: $r_5 \leftarrow r_5 + r_4 \times x_1$.
- Inverse is $r_5 \leftarrow r_5 - r_4 \times x_1$.

Notation: τ_j denotes the starting value of register r_j.

Definition

A sequence of reversible instructions *cleanly computes* f into r_i if, once it finishes:

- $r_i = \tau_i + f(x_1, \ldots, x_n)$
- all other registers are unchanged ($r_j = \tau_j$ for $j \neq i$)


Ring R
Inputs $x_1, \dots, x_n \in R$
Work registers $r_1, \dots, r_m \in R$

Reversible instructions:

- Example: $r_5 \leftarrow r_5 + r_4 \times x_1$.
- Inverse is $r_5 \leftarrow r_5 - r_4 \times x_1$.

Notation: τ_j denotes the starting value of register r_j.

Definition

A sequence of reversible instructions *cleanly computes* f into r_i if, once it finishes:

- $r_i = \tau_i + f(x_1, \dots, x_n)$
- all other registers are unchanged ($r_j = \tau_j$ for $j \neq i$)

A sequence of reversible instructions *cleanly computes* a function f into register i if, once the computation finishes, the new value of register i is its old value τ_i plus f, and every other register is unchanged: r j equals tau j for j not equal to i. Note that we're allowed to use these other registers, as long we make sure to undo all our changes.

Ring R
Inputs $x_1, \ldots, x_n \in R$
Work registers $r_1, \ldots, r_m \in R$

Reversible instructions:

- Example: $r_5 \leftarrow r_5 + r_4 \times x_1$.
- Inverse is $r_5 \leftarrow r_5 - r_4 \times x_1$.

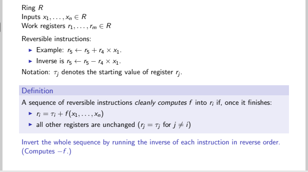
Notation: τ_j denotes the starting value of register r_j.

Definition

A sequence of reversible instructions *cleanly computes f* into r_i if, once it finishes:

- $r_i = \tau_i + f(x_1, \ldots, x_n)$
- all other registers are unchanged ($r_j = \tau_j$ for $j \neq i$)

Invert the whole sequence by running the inverse of each instruction in reverse order. (Computes $-f$.)



Ring R
Inputs $x_1, \dots, x_n \in R$
Work registers $r_1, \dots, r_m \in R$

Reversible instructions:

- Example: $r_5 \leftarrow r_5 + r_4 \times x_1$.
- Inverse is $r_5 \leftarrow r_5 - r_4 \times x_1$.

Notation: τ_j denotes the starting value of register r_j.

Definition

A sequence of reversible instructions *cleanly computes* f into r_i if, once it finishes:

- $r_i = \tau_i + f(x_1, \dots, x_n)$
- all other registers are unchanged ($r_j = \tau_j$ for $j \neq i$)

Invert the whole sequence by running the inverse of each instruction in reverse order. (Computes $-f$.)

Since each instruction is reversible, we can reverse the entire sequence by running the inverses of the original instructions in reverse order. If we do that, the result is a clean computation of negative f.
There are two reasons we like this definition. The first is that it's designed to help us re-use memory, as we'll see later. The second reason is we can translate register programs into branching programs.

Ring R
Inputs $x_1, \ldots, x_n \in R$
Work registers $r_1, \ldots, r_m \in R$

Reversible instructions:

- Example: $r_5 \leftarrow r_5 + r_4 \times x_1$.
- Inverse is $r_5 \leftarrow r_5 - r_4 \times x_1$.

Notation: τ_j denotes the starting value of register r_j.

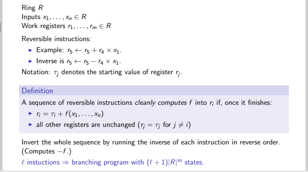
Definition

A sequence of reversible instructions *cleanly computes* f into r_i if, once it finishes:

- $r_i = \tau_i + f(x_1, \ldots, x_n)$
- all other registers are unchanged ($r_j = \tau_j$ for $j \neq i$)

Invert the whole sequence by running the inverse of each instruction in reverse order. (Computes $-f$.)

ℓ instuctions $\Rightarrow$ branching program with $(\ell + 1)|R|^m$ states.



Ring R
Inputs $x_1, \ldots, x_n \in R$
Work registers $r_1, \ldots, r_m \in R$

Reversible instructions:

- Example: $r_5 \leftarrow r_5 + r_4 \times x_1$.
- Inverse is $r_5 \leftarrow r_5 - r_4 \times x_1$.

Notation: τ_j denotes the starting value of register r_j.

Definition

A sequence of reversible instructions *cleanly computes* f into r_i if, once it finishes:

- $r_i = \tau_i + f(x_1, \ldots, x_n)$
- all other registers are unchanged ($r_j = \tau_j$ for $j \neq i$)

Invert the whole sequence by running the inverse of each instruction in reverse order. (Computes $-f$.)

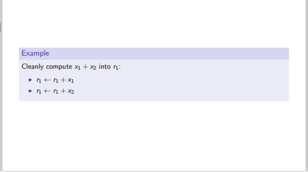
ℓ instuctions $\Rightarrow$ branching program with $(\ell + 1)|R|^m$ states.

If we can cleanly compute f with m registers and ℓ instructions, then we can turn that into a branching program with ℓ plus one layers, each containing R to the m states in order to remember all the register values. So, we can design our algorithm using register instructions and then convert it to a branching program. Now, let's try some examples of clean computation.

Example

Cleanly compute $x_1 + x_2$ into r_1:

- $r_1 \leftarrow r_1 + x_1$
- $r_1 \leftarrow r_1 + x_2$



Example

Cleanly compute $x_1 + x_2$ into r_1:

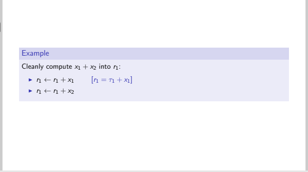
- $r_1 \leftarrow r_1 + x_1$
- $r_1 \leftarrow r_1 + x_2$

For our first example, suppose we want to cleanly compute x one plus x two into register one. We can do this with two instructions: first add x one, then add x two.

Example

Cleanly compute $x_1 + x_2$ into r_1:

- $r_1 \leftarrow r_1 + x_1$ $\qquad [r_1 = \tau_1 + x_1]$
- $r_1 \leftarrow r_1 + x_2$



Example

Cleanly compute $x_1 + x_2$ into r_1:

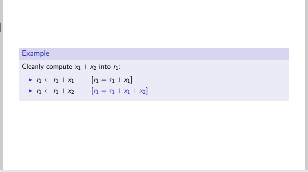
- $r_1 \leftarrow r_1 + x_1$ $[r_1 = \tau_1 + x_1]$
- $r_1 \leftarrow r_1 + x_2$

After we add x one, the value of the register is tau one plus x one.

Example

Cleanly compute $x_1 + x_2$ into r_1:

- $r_1 \leftarrow r_1 + x_1$ $[r_1 = \tau_1 + x_1]$
- $r_1 \leftarrow r_1 + x_2$ $[r_1 = \tau_1 + x_1 + x_2]$



Example

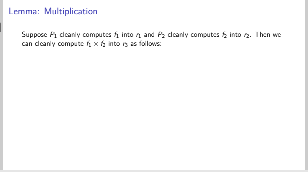
Cleanly compute $x_1 + x_2$ into r_1:

- $r_1 \leftarrow r_1 + x_1$ $[r_1 = \tau_1 + x_1]$
- $r_1 \leftarrow r_1 + x_2$ $[r_1 = \tau_1 + x_1 + x_2]$

And after we add x two, the value of the register is tau one plus x one plus x two. By definition, we've cleanly computed x1 plus x2 into r1.

Lemma: Multiplication

Suppose P_1 cleanly computes f_1 into r_1 and P_2 cleanly computes f_2 into r_2. Then we can cleanly compute $f_1 \times f_2$ into r_3 as follows:



Lemma: Multiplication

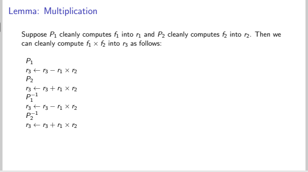
Suppose P_1 cleanly computes f_1 into r_1 and P_2 cleanly computes f_2 into r_2. Then we can cleanly compute $f_1 \times f_2$ into r_3 as follows:

For our next example, let's say we've got a subroutine P one that cleanly computes a function f 1, and a subroutine P two that cleanly computes a function f 2, and our goal is to compute the product f 1 times f 2.

Lemma: Multiplication

Suppose P_1 cleanly computes f_1 into r_1 and P_2 cleanly computes f_2 into r_2. Then we can cleanly compute $f_1 \times f_2$ into r_3 as follows:

```
$P_1$
$r_3 \leftarrow r_3 - r_1 \times r_2$
$P_2$
$r_3 \leftarrow r_3 + r_1 \times r_2$
$P_1^{-1}$
$r_3 \leftarrow r_3 - r_1 \times r_2$
$P_2^{-1}$
$r_3 \leftarrow r_3 + r_1 \times r_2$
```



Lemma: Multiplication

Suppose P_1 cleanly computes f_1 into r_1 and P_2 cleanly computes f_2 into r_2. Then we can cleanly compute $f_1 \times f_2$ into r_3 as follows:

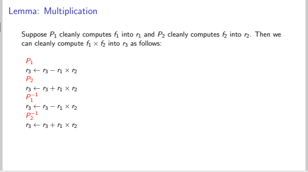
```
$P_1$
$r_3 \leftarrow r_3 - r_1 \times r_2$
$P_2$
$r_3 \leftarrow r_3 + r_1 \times r_2$
$P_1^{-1}$
$r_3 \leftarrow r_3 - r_1 \times r_2$
$P_2^{-1}$
$r_3 \leftarrow r_3 + r_1 \times r_2$
```

The program looks like this. We can think of it as being made out of two interlocking pieces.

Lemma: Multiplication

Suppose P_1 cleanly computes f_1 into r_1 and P_2 cleanly computes f_2 into r_2. Then we can cleanly compute $f_1 \times f_2$ into r_3 as follows:

```
$P_1$
$r_3 \leftarrow r_3 - r_1 \times r_2$
$P_2$
$r_3 \leftarrow r_3 + r_1 \times r_2$
$P_1^{-1}$
$r_3 \leftarrow r_3 - r_1 \times r_2$
$P_2^{-1}$
$r_3 \leftarrow r_3 + r_1 \times r_2$
```



Lemma: Multiplication

Suppose P_1 cleanly computes f_1 into r_1 and P_2 cleanly computes f_2 into r_2. Then we can cleanly compute $f_1 \times f_2$ into r_3 as follows:

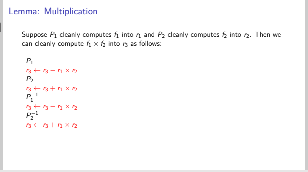
```
$P_1$
$r_3 \leftarrow r_3 - r_1 \times r_2$
$P_2$
$r_3 \leftarrow r_3 + r_1 \times r_2$
$P_1^{-1}$
$r_3 \leftarrow r_3 - r_1 \times r_2$
$P_2^{-1}$
$r_3 \leftarrow r_3 + r_1 \times r_2$
```

The first piece is calling the subroutines P one and P two. We first call P one, then P two. Since everything's made out of reversible instructions, we're then able to run P one backward and P two backward.

Lemma: Multiplication

Suppose P_1 cleanly computes f_1 into r_1 and P_2 cleanly computes f_2 into r_2. Then we can cleanly compute $f_1 \times f_2$ into r_3 as follows:

```
$P_1$
$r_3 \leftarrow r_3 - r_1 \times r_2$
$P_2$
$r_3 \leftarrow r_3 + r_1 \times r_2$
$P_1^{-1}$
$r_3 \leftarrow r_3 - r_1 \times r_2$
$P_2^{-1}$
$r_3 \leftarrow r_3 + r_1 \times r_2$
```



Lemma: Multiplication

Suppose P_1 cleanly computes f_1 into r_1 and P_2 cleanly computes f_2 into r_2. Then we can cleanly compute $f_1 \times f_2$ into r_3 as follows:

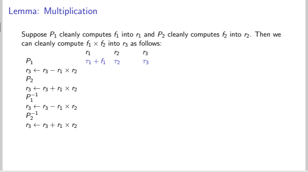
```
$P_1$
$r_3 \leftarrow r_3 - r_1 \times r_2$
$P_2$
$r_3 \leftarrow r_3 + r_1 \times r_2$
$P_1^{-1}$
$r_3 \leftarrow r_3 - r_1 \times r_2$
$P_2^{-1}$
$r_3 \leftarrow r_3 + r_1 \times r_2$
```

The other piece is adding and subtracting r one times r two. Since the subroutines are modifying the contents of r one and r two, this has a different effect each time. So, let's see what happens when we run the program.

Lemma: Multiplication

Suppose P_1 cleanly computes f_1 into r_1 and P_2 cleanly computes f_2 into r_2. Then we can cleanly compute $f_1 \times f_2$ into r_3 as follows:

	r_1	r_2	r_3
P_1	$\tau_1 + f_1$	τ_2	τ_3
$r_3 \leftarrow r_3 - r_1 \times r_2$			
P_2			
$r_3 \leftarrow r_3 + r_1 \times r_2$			
P_1^{-1}			
$r_3 \leftarrow r_3 - r_1 \times r_2$			
P_2^{-1}			
$r_3 \leftarrow r_3 + r_1 \times r_2$			



Lemma: Multiplication

Suppose P_1 cleanly computes f_1 into r_1 and P_2 cleanly computes f_2 into r_2. Then we can cleanly compute $f_1 \times f_2$ into r_3 as follows:

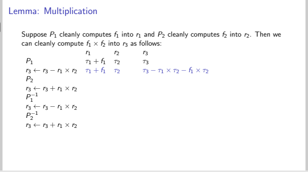
	r_1	r_2	r_3
P_1	$\tau_1 + f_1$	τ_2	τ_3
$r_3 \leftarrow r_3 - r_1 \times r_2$			
P_2			
$r_3 \leftarrow r_3 + r_1 \times r_2$			
P_1^{-1}			
$r_3 \leftarrow r_3 - r_1 \times r_2$			
P_2^{-1}			
$r_3 \leftarrow r_3 + r_1 \times r_2$			

We start by running P1. After it's finished, register one has value tau 1 plus f 1, and the other two registers have their original values tau 2 and tau 3.

Lemma: Multiplication

Suppose P_1 cleanly computes f_1 into r_1 and P_2 cleanly computes f_2 into r_2. Then we can cleanly compute $f_1 \times f_2$ into r_3 as follows:

	r_1	r_2	r_3
P_1	$\tau_1 + f_1$	τ_2	τ_3
$r_3 \leftarrow r_3 - r_1 \times r_2$	$\tau_1 + f_1$	τ_2	$\tau_3 - \tau_1 \times \tau_2 - f_1 \times \tau_2$
P_2			
$r_3 \leftarrow r_3 + r_1 \times r_2$			
P_1^{-1}			
$r_3 \leftarrow r_3 - r_1 \times r_2$			
P_2^{-1}			
$r_3 \leftarrow r_3 + r_1 \times r_2$			



Lemma: Multiplication

Suppose P_1 cleanly computes f_1 into r_1 and P_2 cleanly computes f_2 into r_2. Then we can cleanly compute $f_1 \times f_2$ into r_3 as follows:

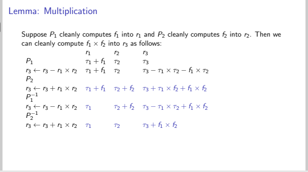
	r_1	r_2	r_3
P_1	$\tau_1 + f_1$	τ_2	τ_3
$r_3 \leftarrow r_3 - r_1 \times r_2$	$\tau_1 + f_1$	τ_2	$\tau_3 - \tau_1 \times \tau_2 - f_1 \times \tau_2$
P_2			
$r_3 \leftarrow r_3 + r_1 \times r_2$			
P_1^{-1}			
$r_3 \leftarrow r_3 - r_1 \times r_2$			
P_2^{-1}			
$r_3 \leftarrow r_3 + r_1 \times r_2$			

The next instruction subtracts two terms from register three, leaving a value of tau 3 minus tau 1 times tau 2 minus f one times tau 2.

Lemma: Multiplication

Suppose P_1 cleanly computes f_1 into r_1 and P_2 cleanly computes f_2 into r_2. Then we can cleanly compute $f_1 \times f_2$ into r_3 as follows:

	r_1	r_2	r_3
P_1	$\tau_1 + f_1$	τ_2	τ_3
$r_3 \leftarrow r_3 - r_1 \times r_2$	$\tau_1 + f_1$	τ_2	$\tau_3 - \tau_1 \times \tau_2 - f_1 \times \tau_2$
P_2			
$r_3 \leftarrow r_3 + r_1 \times r_2$	$\tau_1 + f_1$	$\tau_2 + f_2$	$\tau_3 + \tau_1 \times f_2 + f_1 \times f_2$
P_1^{-1}			
$r_3 \leftarrow r_3 - r_1 \times r_2$	τ_1	$\tau_2 + f_2$	$\tau_3 - \tau_1 \times \tau_2 + f_1 \times f_2$
P_2^{-1}			
$r_3 \leftarrow r_3 + r_1 \times r_2$	τ_1	τ_2	$\tau_3 + f_1 \times f_2$



Lemma: Multiplication

Suppose P_1 cleanly computes f_1 into r_1 and P_2 cleanly computes f_2 into r_2. Then we can cleanly compute $f_1 \times f_2$ into r_3 as follows:

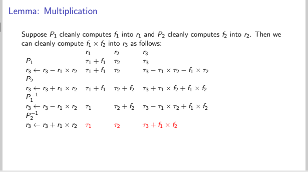
	r_1	r_2	r_3
P_1	$\tau_1 + f_1$	τ_2	τ_3
$r_3 \leftarrow r_3 - r_1 \times r_2$	$\tau_1 + f_1$	τ_2	$\tau_3 - \tau_1 \times \tau_2 - f_1 \times \tau_2$
P_2			
$r_3 \leftarrow r_3 + r_1 \times r_2$	$\tau_1 + f_1$	$\tau_2 + f_2$	$\tau_3 + \tau_1 \times f_2 + f_1 \times f_2$
P_1^{-1}			
$r_3 \leftarrow r_3 - r_1 \times r_2$	τ_1	$\tau_2 + f_2$	$\tau_3 - \tau_1 \times \tau_2 + f_1 \times f_2$
P_2^{-1}			
$r_3 \leftarrow r_3 + r_1 \times r_2$	τ_1	τ_2	$\tau_3 + f_1 \times f_2$

As the program continues, different terms are added and subtracted from register 3.

Lemma: Multiplication

Suppose P_1 cleanly computes f_1 into r_1 and P_2 cleanly computes f_2 into r_2. Then we can cleanly compute $f_1 \times f_2$ into r_3 as follows:

	r_1	r_2	r_3
P_1	$\tau_1 + f_1$	τ_2	τ_3
$r_3 \leftarrow r_3 - r_1 \times r_2$	$\tau_1 + f_1$	τ_2	$\tau_3 - \tau_1 \times \tau_2 - f_1 \times \tau_2$
P_2			
$r_3 \leftarrow r_3 + r_1 \times r_2$	$\tau_1 + f_1$	$\tau_2 + f_2$	$\tau_3 + \tau_1 \times f_2 + f_1 \times f_2$
P_1^{-1}			
$r_3 \leftarrow r_3 - r_1 \times r_2$	τ_1	$\tau_2 + f_2$	$\tau_3 - \tau_1 \times \tau_2 + f_1 \times f_2$
P_2^{-1}			
$r_3 \leftarrow r_3 + r_1 \times r_2$	τ_1	τ_2	$\tau_3 + f_1 \times f_2$



Lemma: Multiplication

Suppose P_1 cleanly computes f_1 into r_1 and P_2 cleanly computes f_2 into r_2. Then we can cleanly compute $f_1 \times f_2$ into r_3 as follows:

	r_1	r_2	r_3
P_1	$\tau_1 + f_1$	τ_2	τ_3
$r_3 \leftarrow r_3 - r_1 \times r_2$	$\tau_1 + f_1$	τ_2	$\tau_3 - \tau_1 \times \tau_2 - f_1 \times \tau_2$
P_2			
$r_3 \leftarrow r_3 + r_1 \times r_2$	$\tau_1 + f_1$	$\tau_2 + f_2$	$\tau_3 + \tau_1 \times f_2 + f_1 \times f_2$
P_1^{-1}			
$r_3 \leftarrow r_3 - r_1 \times r_2$	τ_1	$\tau_2 + f_2$	$\tau_3 - \tau_1 \times \tau_2 + f_1 \times f_2$
P_2^{-1}			
$r_3 \leftarrow r_3 + r_1 \times r_2$	τ_1	τ_2	$\tau_3 + f_1 \times f_2$

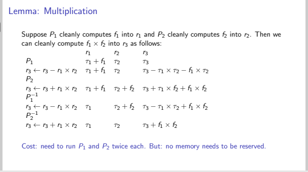
At the end, register three holds its original value plus f1 times f2, and the other registers have been restored. That means we've succeeded.

Lemma: Multiplication

Suppose P_1 cleanly computes f_1 into r_1 and P_2 cleanly computes f_2 into r_2. Then we can cleanly compute $f_1 \times f_2$ into r_3 as follows:

	r_1	r_2	r_3
P_1	$\tau_1 + f_1$	τ_2	τ_3
$r_3 \leftarrow r_3 - r_1 \times r_2$	$\tau_1 + f_1$	τ_2	$\tau_3 - \tau_1 \times \tau_2 - f_1 \times \tau_2$
P_2			
$r_3 \leftarrow r_3 + r_1 \times r_2$	$\tau_1 + f_1$	$\tau_2 + f_2$	$\tau_3 + \tau_1 \times f_2 + f_1 \times f_2$
P_1^{-1}			
$r_3 \leftarrow r_3 - r_1 \times r_2$	τ_1	$\tau_2 + f_2$	$\tau_3 - \tau_1 \times \tau_2 + f_1 \times f_2$
P_2^{-1}			
$r_3 \leftarrow r_3 + r_1 \times r_2$	τ_1	τ_2	$\tau_3 + f_1 \times f_2$

Cost: need to run P_1 and P_2 twice each. But: no memory needs to be reserved.



Lemma: Multiplication

Suppose P_1 cleanly computes f_1 into r_1 and P_2 cleanly computes f_2 into r_2. Then we can cleanly compute $f_1 \times f_2$ into r_3 as follows:

	r_1	r_2	r_3
P_1	$\tau_1 + f_1$	τ_2	τ_3
$r_3 \leftarrow r_3 - r_1 \times r_2$	$\tau_1 + f_1$	τ_2	$\tau_3 - \tau_1 \times \tau_2 - f_1 \times \tau_2$
P_2			
$r_3 \leftarrow r_3 + r_1 \times r_2$	$\tau_1 + f_1$	$\tau_2 + f_2$	$\tau_3 + \tau_1 \times f_2 + f_1 \times f_2$
P_1^{-1}			
$r_3 \leftarrow r_3 - r_1 \times r_2$	τ_1	$\tau_2 + f_2$	$\tau_3 - \tau_1 \times \tau_2 + f_1 \times f_2$
P_2^{-1}			
$r_3 \leftarrow r_3 + r_1 \times r_2$	τ_1	τ_2	$\tau_3 + f_1 \times f_2$

Cost: need to run P_1 and P_2 twice each. But: no memory needs to be reserved.

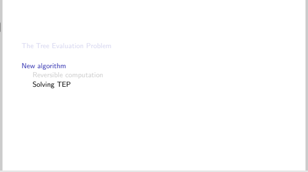
Now, we've computed f1 times f2, but what did it cost us? Well, we had to make four subroutine calls: P1 and P2 forward and backward. But, this algorithm is extremely efficient with memory. Notice that P1 and P2 are allowed to use all of our memory, as long as they restore it when they're done. There is no memory that needs to be set aside for the parent routine's exclusive use. I like to think of these programs as "borrowing" the memory they use.
Now let's talk about how to apply these techniques to solving the Tree Evaluation Problem.

The Tree Evaluation Problem

New algorithm

Reversible computation

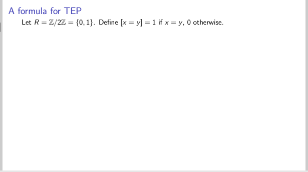
Solving TEP



This is the last part of the video.
We want to build a reversible computation to compute the value at the root node of the tree. In order to do that, it will be helpful to have an algebraic formula for that root value.

A formula for TEP

Let $R = \mathbb{Z}/2\mathbb{Z} = \{0, 1\}$. Define $[x = y] = 1$ if $x = y$, 0 otherwise.



A formula for TEP

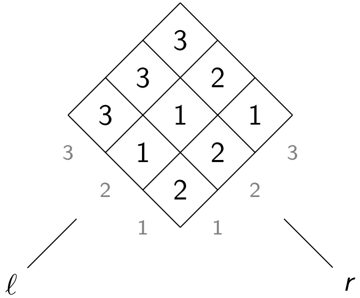
Let $R = \mathbb{Z}/2\mathbb{Z} = \{0, 1\}$. Define $[x = y] = 1$ if $x = y$, 0 otherwise.

From here on, our ring will be the integers mod two, meaning registers will store bits. I'll introduce some notation: brackets x equals y is an *indicator* which equals one if they are equal and otherwise zero.

A formula for TEP

Let $R = \mathbb{Z}/2\mathbb{Z} = \{0, 1\}$. Define $[x = y] = 1$ if $x = y$, 0 otherwise.

Suppose node v has children ℓ and r:

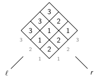


$[v = 1] =$

A formula for TEP

Let $R = \mathbb{Z}/2\mathbb{Z} = \{0, 1\}$. Define $[x = y] = 1$ if $x = y$, 0 otherwise.

Suppose node v has children ℓ and r:



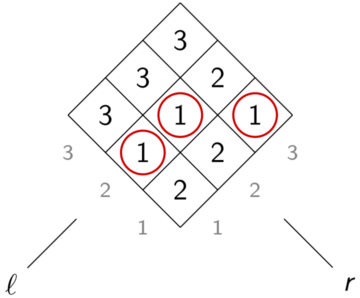
$[v = 1] =$

Now, suppose we have some node v with two children, ℓ and r, and this is the table at that node. Let's try to build a formula for the indicater v equals one.

A formula for TEP

Let $R = \mathbb{Z}/2\mathbb{Z} = \{0, 1\}$. Define $[x = y] = 1$ if $x = y$, 0 otherwise.

Suppose node v has children ℓ and r:

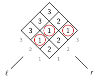


$[v = 1] =$

A formula for TEP

Let $R = \mathbb{Z}/2\mathbb{Z} = \{0, 1\}$. Define $[x = y] = 1$ if $x = y$, 0 otherwise.

Suppose node v has children ℓ and r:



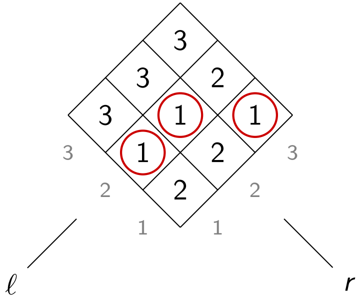
$[v = 1] =$

Well, there are three ways that node v can be equal to one, corresponding to the three times one appears in the table at node v. We can turn this into a formula with three terms.

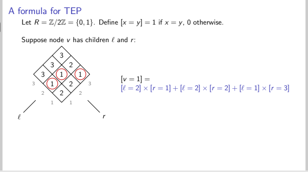
A formula for TEP

Let $R = \mathbb{Z}/2\mathbb{Z} = \{0, 1\}$. Define $[x = y] = 1$ if $x = y$, 0 otherwise.

Suppose node v has children ℓ and r:



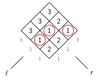
$$[v = 1] =$$
$$[\ell = 2] \times [r = 1] + [\ell = 2] \times [r = 2] + [\ell = 1] \times [r = 3]$$



A formula for TEP

Let $R = \mathbb{Z}/2\mathbb{Z} = \{0, 1\}$. Define $[x = y] = 1$ if $x = y$, 0 otherwise.

Suppose node v has children ℓ and r:



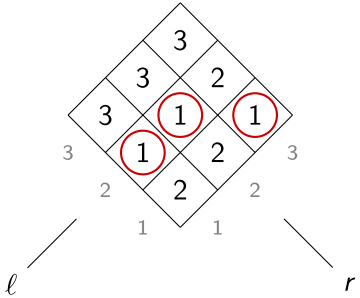
$[v = 1] =$
$[\ell = 2] \times [r = 1] + [\ell = 2] \times [r = 2] + [\ell = 1] \times [r = 3]$

The terms say: either ℓ equals 2 and r equals 1, or ℓ equals 2 and r equals 2, or ℓ equals 1 and r equals 3.
Now let's write the general formula.

A formula for TEP

Let $R = \mathbb{Z}/2\mathbb{Z} = \{0, 1\}$. Define $[x = y] = 1$ if $x = y$, 0 otherwise.

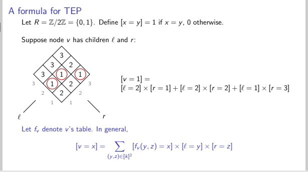
Suppose node v has children ℓ and r:



$$[v = 1] =$$
$$[\ell = 2] \times [r = 1] + [\ell = 2] \times [r = 2] + [\ell = 1] \times [r = 3]$$

Let f_v denote v's table. In general,

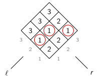
$$[v = x] = \sum_{(y,z)\in[k]^2} [f_v(y, z) = x] \times [\ell = y] \times [r = z]$$



A formula for TEP

Let $R = \mathbb{Z}/2\mathbb{Z} = \{0, 1\}$. Define $[x = y] = 1$ if $x = y$, 0 otherwise.

Suppose node v has children ℓ and r:



$[v = 1] =$
$[\ell = 2] \times [r = 1] + [\ell = 2] \times [r = 2] + [\ell = 1] \times [r = 3]$

Let f_v denote v's table. In general,

$$[v = x] = \sum_{(y,z)\in[k]^2} [f_v(y, z) = x] \times [\ell = y] \times [r = z]$$

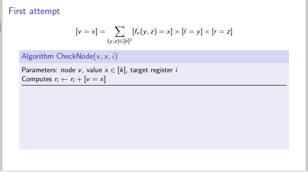
Let's say f v is the table of values at node v.
In general, we take the sum over all possible values y and z for the two children. Inside the sum, we check node v's table to see whether each term should be included. We multiply that indicator by the indicators ℓ equals y and r equals z.
With that formula in hand, let's try to build a recursive algorithm.

First attempt

$$[v = x] = \sum_{(y,z)\in[k]^2} [f_v(y,z) = x] \times [\ell = y] \times [r = z]$$

Algorithm CheckNode(v, x, i)

Parameters: node v, value $x \in [k]$, target register i
Computes $r_i \leftarrow r_i + [v = x]$



First attempt

$$[v = x] = \sum_{(y,z)\in[k]^2} [f_v(y,z) = x] \times [\ell = y] \times [r = z]$$

Algorithm CheckNode(v, x, i)

Parameters: node v, value $x \in [k]$, target register i
Computes $r_i \leftarrow r_i + [v = x]$

I've left our formula at the top of the slide for reference. Our algorithm's goal is to compute the formula, which determines whether node v has value x.
The algorithm is parameterized by the node v, the value x, and some target register i. If node v has value x, it will flip the bit in register i. In other words, it assigns r i plus the indicator v equals x to r i.

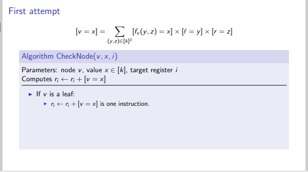
First attempt

$$[v = x] = \sum_{(y,z)\in[k]^2} [f_v(y,z) = x] \times [\ell = y] \times [r = z]$$

Algorithm CheckNode(v, x, i)

Parameters: node v, value $x \in [k]$, target register i
Computes $r_i \leftarrow r_i + [v = x]$

- If v is a leaf:
    - $r_i \leftarrow r_i + [v = x]$ is one instruction.



First attempt

$$[v = x] = \sum_{(y,z)\in[k]^2} [f_v(y,z) = x] \times [\ell = y] \times [r = z]$$

Algorithm CheckNode(v, x, i)

Parameters: node v, value $x \in [k]$, target register i
Computes $r_i \leftarrow r_i + [v = x]$

- If v is a leaf:
  - $r_i \leftarrow r_i + [v = x]$ is one instruction.

If v is a leaf node, then the value of v is directly available as part of the input. So, we can do this in just one instruction.

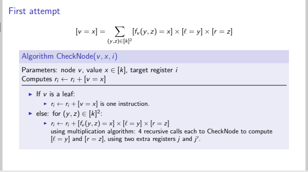
First attempt

$$[v = x] = \sum_{(y,z)\in[k]^2} [f_v(y,z) = x] \times [\ell = y] \times [r = z]$$

Algorithm CheckNode(v, x, i)

Parameters: node v, value $x \in [k]$, target register i
Computes $r_i \leftarrow r_i + [v = x]$

- If v is a leaf:
  - $r_i \leftarrow r_i + [v = x]$ is one instruction.
- else: for $(y, z) \in [k]^2$:
  - $r_i \leftarrow r_i + [f_v(y,z) = x] \times [\ell = y] \times [r = z]$
    using multiplication algorithm: 4 recursive calls each to CheckNode to compute $[\ell = y]$ and $[r = z]$, using two extra registers j and j'.



First attempt

$$[v = x] = \sum_{(y,z)\in[k]^2} [f_v(y,z) = x] \times [\ell = y] \times [r = z]$$

Algorithm CheckNode(v, x, i)

Parameters: node v, value $x \in [k]$, target register i
Computes $r_i \leftarrow r_i + [v = x]$

```
- If $v$ is a leaf:
    - $r_i \leftarrow r_i + [v = x]$ is one instruction.
- else: for $(y, z) \in [k]^2$:
    - $r_i \leftarrow r_i + [f_v(y,z) = x] \times [\ell = y] \times [r = z]$
      using multiplication algorithm: 4 recursive calls each to CheckNode to compute
      $[\ell = y]$ and $[r = z]$, using two extra registers $j$ and $j'$.
```

If v is an internal node, then we compute this formula by looping over all k squared possible values for y and z and adding each term to r i one at a time.

Each term includes a product of the indicators l equals y times r equals z, which we compute using the multiplication algorithm. This requires four recursive calls to CheckNode and two auxiliary registers j and j'.

First attempt

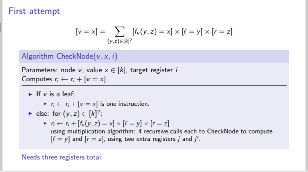
$$[v = x] = \sum_{(y,z)\in[k]^2} [f_v(y,z) = x] \times [\ell = y] \times [r = z]$$

Algorithm CheckNode(v, x, i)

Parameters: node v, value $x \in [k]$, target register i
Computes $r_i \leftarrow r_i + [v = x]$

- If v is a leaf:
  - $r_i \leftarrow r_i + [v = x]$ is one instruction.
- else: for $(y, z) \in [k]^2$:
  - $r_i \leftarrow r_i + [f_v(y,z) = x] \times [\ell = y] \times [r = z]$
    using multiplication algorithm: 4 recursive calls each to CheckNode to compute $[\ell = y]$ and $[r = z]$, using two extra registers j and j'.

Needs three registers total.



First attempt

$$[v = x] = \sum_{(y,z)\in[k]^2} [f_v(y,z) = x] \times [\ell = y] \times [r = z]$$

Algorithm CheckNode(v, x, i)

Parameters: node v, value $x \in [k]$, target register i
Computes $r_i \leftarrow r_i + [v = x]$

- If v is a leaf:
  - $r_i \leftarrow r_i + [v = x]$ is one instruction.
- else: for $(y, z) \in [k]^2$:
  - $r_i \leftarrow r_i + [f_v(y,z) = x] \times [\ell = y] \times [r = z]$
    using multiplication algorithm: 4 recursive calls each to CheckNode to compute $[\ell = y]$ and $[r = z]$, using two extra registers j and j'.

Needs three registers total.

We use a total of three registers: register i holds our output, and two more registers j and j' are required by the multiplication algorithm. Since we're using clean computations, the calls to the subroutine are free to use those same three registers, so we really don't need any more than three registers, including all the recursive calls.

First attempt

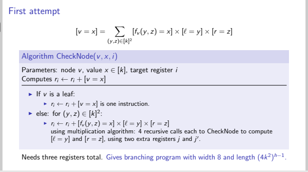
$$[v = x] = \sum_{(y,z)\in[k]^2} [f_v(y,z) = x] \times [\ell = y] \times [r = z]$$

Algorithm CheckNode(v, x, i)

Parameters: node v, value $x \in [k]$, target register i
Computes $r_i \leftarrow r_i + [v = x]$

- If v is a leaf:
  - $r_i \leftarrow r_i + [v = x]$ is one instruction.
- else: for $(y,z) \in [k]^2$:
  - $r_i \leftarrow r_i + [f_v(y,z) = x] \times [\ell = y] \times [r = z]$ using multiplication algorithm: 4 recursive calls each to CheckNode to compute $[\ell = y]$ and $[r = z]$, using two extra registers j and j'.

Needs three registers total. Gives branching program with width 8 and length $(4k^2)^{h-1}$.



First attempt

$$[v = x] = \sum_{(y,z)\in[k]^2} [f_v(y,z) = x] \times [\ell = y] \times [r = z]$$

Algorithm CheckNode(v, x, i)

```
Parameters: node $v$, value $x \in [k]$, target register $i$
Computes $r_i \leftarrow r_i + [v = x]$
  ▸ If $v$ is a leaf:
      ▸ $r_i \leftarrow r_i + [v = x]$ is one instruction.
  ▸ else: for $(y, z) \in [k]^2$:
      ▸ $r_i \leftarrow r_i + [f_v(y,z) = x] \times [\ell = y] \times [r = z]$
        using multiplication algorithm: 4 recursive calls each to CheckNode to compute
        $[\ell = y]$ and $[r = z]$, using two extra registers $j$ and $j'$.
```

Needs three registers total. Gives branching program with width 8 and length $(4k^2)^{h-1}$.

If we convert this to a branching program, those three one-bit registers translate to eight states in each layer. The length of the program is four k squared to the power h minus one, since at every level, we make four k squared recursive calls.
This isn't very good.

First attempt

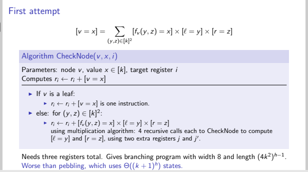
$$[v = x] = \sum_{(y,z)\in[k]^2} [f_v(y,z) = x] \times [\ell = y] \times [r = z]$$

Algorithm CheckNode(v, x, i)

Parameters: node v, value $x \in [k]$, target register i
Computes $r_i \leftarrow r_i + [v = x]$

- If v is a leaf:
  - $r_i \leftarrow r_i + [v = x]$ is one instruction.
- else: for $(y, z) \in [k]^2$:
  - $r_i \leftarrow r_i + [f_v(y,z) = x] \times [\ell = y] \times [r = z]$
    using multiplication algorithm: 4 recursive calls each to CheckNode to compute $[\ell = y]$ and $[r = z]$, using two extra registers j and j'.

Needs three registers total. Gives branching program with width 8 and length $(4k^2)^{h-1}$.
Worse than pebbling, which uses $\Theta((k+1)^h)$ states.



First attempt

$$[v = x] = \sum_{(y,z)\in[k]^2} [f_v(y,z) = x] \times [\ell = y] \times [r = z]$$

Algorithm CheckNode(v, x, i)

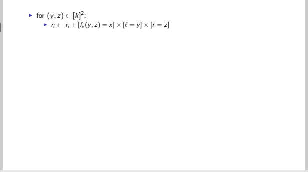
Parameters: node v, value $x \in [k]$, target register i
Computes $r_i \leftarrow r_i + [v = x]$

```
▸ If $v$ is a leaf:
    ▸ $r_i \leftarrow r_i + [v = x]$ is one instruction.
▸ else: for $(y, z) \in [k]^2$:
    ▸ $r_i \leftarrow r_i + [f_v(y,z) = x] \times [\ell = y] \times [r = z]$
      using multiplication algorithm: 4 recursive calls each to CheckNode to compute
      $[\ell = y]$ and $[r = z]$, using two extra registers $j$ and $j'$.
```

Needs three registers total. Gives branching program with width 8 and length $(4k^2)^{h-1}$.
Worse than pebbling, which uses $\Theta((k+1)^h)$ states.

Our original pebbling algorithm just uses k plus one to the h states. So, we'll need another trick if we're going to beat it.
Let's take a closer look at what's going on in this for loop.

- for $(y, z) \in [k]^2$:
    - $r_i \leftarrow r_i + [f_v(y, z) = x] \times [\ell = y] \times [r = z]$



- for $(y, z) \in [k]^2$:
  - $r_i \leftarrow r_i + [f_v(y, z) = x] \times [\ell = y] \times [r = z]$

Each iteration of the for loop is using the multiplication lemma to combine the indicators ℓ equals y and r equals z.

```
for $(y, z) \in [k]^2$:
    $r_i \leftarrow r_i + [f_v(y,z) = x] \times [\ell = y] \times [r = z]$
```

$r_j \leftarrow r_j + [\ell = 1]$

$r_i \leftarrow r_i - r_j \times r_{j'}$

$r_{j'} \leftarrow r_{j'} + [r = 1]$

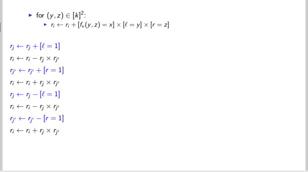
$r_i \leftarrow r_i + r_j \times r_{j'}$

$r_j \leftarrow r_j - [\ell = 1]$

$r_i \leftarrow r_i - r_j \times r_{j'}$

$r_{j'} \leftarrow r_{j'} - [r = 1]$

$r_i \leftarrow r_i + r_j \times r_{j'}$



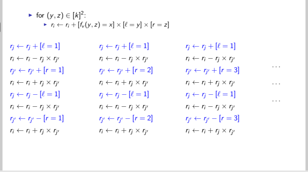
- for $(y, z) \in [k]^2$:
    - $r_i \leftarrow r_i + [f_v(y, z) = x] \times [\ell = y] \times [r = z]$

```
$r_j \leftarrow r_j + [\ell = 1]$
$r_i \leftarrow r_i - r_j \times r_{j'}$
$r_{j'} \leftarrow r_{j'} + [r = 1]$
$r_i \leftarrow r_i + r_j \times r_{j'}$
$r_j \leftarrow r_j - [\ell = 1]$
$r_i \leftarrow r_i - r_j \times r_{j'}$
$r_{j'} \leftarrow r_{j'} - [r = 1]$
$r_i \leftarrow r_i + r_j \times r_{j'}$
```

If you remember the multiplication lemma, it looks kind of like this. We make four calls to our subroutines for checking ℓ and r, and in between those four calls, we update our final output register r i. I've coloured the recursive calls in blue.

```
- for $(y, z) \in [k]^2$:
    - $r_i \leftarrow r_i + [f_v(y, z) = x] \times [\ell = y] \times [r = z]$
```

$$\begin{array}{llll}
r_j \leftarrow r_j + [\ell = 1] & r_j \leftarrow r_j + [\ell = 1] & r_j \leftarrow r_j + [\ell = 1] & \\
r_i \leftarrow r_i - r_j \times r_{j'} & r_i \leftarrow r_i - r_j \times r_{j'} & r_i \leftarrow r_i - r_j \times r_{j'} & \\
r_{j'} \leftarrow r_{j'} + [r = 1] & r_{j'} \leftarrow r_{j'} + [r = 2] & r_{j'} \leftarrow r_{j'} + [r = 3] & \dots \\
r_i \leftarrow r_i + r_j \times r_{j'} & r_i \leftarrow r_i + r_j \times r_{j'} & r_i \leftarrow r_i + r_j \times r_{j'} & \dots \\
r_j \leftarrow r_j - [\ell = 1] & r_j \leftarrow r_j - [\ell = 1] & r_j \leftarrow r_j - [\ell = 1] & \dots \\
r_i \leftarrow r_i - r_j \times r_{j'} & r_i \leftarrow r_i - r_j \times r_{j'} & r_i \leftarrow r_i - r_j \times r_{j'} & \\
r_{j'} \leftarrow r_{j'} - [r = 1] & r_{j'} \leftarrow r_{j'} - [r = 2] & r_{j'} \leftarrow r_{j'} - [r = 3] & \\
r_i \leftarrow r_i + r_j \times r_{j'} & r_i \leftarrow r_i + r_j \times r_{j'} & r_i \leftarrow r_i + r_j \times r_{j'} &
\end{array}$$



- for $(y, z) \in [k]^2$:
  - $r_i \leftarrow r_i + [f_v(y,z) = x] \times [\ell = y] \times [r = z]$

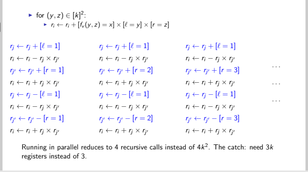
$r_j \leftarrow r_j + [\ell = 1]$	$r_j \leftarrow r_j + [\ell = 1]$	$r_j \leftarrow r_j + [\ell = 1]$	
$r_i \leftarrow r_i - r_j \times r_{j'}$	$r_i \leftarrow r_i - r_j \times r_{j'}$	$r_i \leftarrow r_i - r_j \times r_{j'}$	
$r_{j'} \leftarrow r_{j'} + [r = 1]$	$r_{j'} \leftarrow r_{j'} + [r = 2]$	$r_{j'} \leftarrow r_{j'} + [r = 3]$	$\dots$
$r_i \leftarrow r_i + r_j \times r_{j'}$	$r_i \leftarrow r_i + r_j \times r_{j'}$	$r_i \leftarrow r_i + r_j \times r_{j'}$	$\dots$
$r_j \leftarrow r_j - [\ell = 1]$	$r_j \leftarrow r_j - [\ell = 1]$	$r_j \leftarrow r_j - [\ell = 1]$	$\dots$
$r_i \leftarrow r_i - r_j \times r_{j'}$	$r_i \leftarrow r_i - r_j \times r_{j'}$	$r_i \leftarrow r_i - r_j \times r_{j'}$	
$r_{j'} \leftarrow r_{j'} - [r = 1]$	$r_{j'} \leftarrow r_{j'} - [r = 2]$	$r_{j'} \leftarrow r_{j'} - [r = 3]$	
$r_i \leftarrow r_i + r_j \times r_{j'}$	$r_i \leftarrow r_i + r_j \times r_{j'}$	$r_i \leftarrow r_i + r_j \times r_{j'}$	

The for loop just means we do this whole thing over and over again, k squared times.
It turns out we can completely parallelize this. All of the instructions on the first row can be run at the same time, with one recursive call that checks all of the possible values for the left child. We can do similar things for the other lines.

- for $(y, z) \in [k]^2$:
  - $r_i \leftarrow r_i + [f_v(y,z) = x] \times [\ell = y] \times [r = z]$

$r_j \leftarrow r_j + [\ell = 1]$	$r_j \leftarrow r_j + [\ell = 1]$	$r_j \leftarrow r_j + [\ell = 1]$	
$r_i \leftarrow r_i - r_j \times r_{j'}$	$r_i \leftarrow r_i - r_j \times r_{j'}$	$r_i \leftarrow r_i - r_j \times r_{j'}$	
$r_{j'} \leftarrow r_{j'} + [r = 1]$	$r_{j'} \leftarrow r_{j'} + [r = 2]$	$r_{j'} \leftarrow r_{j'} + [r = 3]$	$\cdots$
$r_i \leftarrow r_i + r_j \times r_{j'}$	$r_i \leftarrow r_i + r_j \times r_{j'}$	$r_i \leftarrow r_i + r_j \times r_{j'}$	$\cdots$
$r_j \leftarrow r_j - [\ell = 1]$	$r_j \leftarrow r_j - [\ell = 1]$	$r_j \leftarrow r_j - [\ell = 1]$	$\cdots$
$r_i \leftarrow r_i - r_j \times r_{j'}$	$r_i \leftarrow r_i - r_j \times r_{j'}$	$r_i \leftarrow r_i - r_j \times r_{j'}$	
$r_{j'} \leftarrow r_{j'} - [r = 1]$	$r_{j'} \leftarrow r_{j'} - [r = 2]$	$r_{j'} \leftarrow r_{j'} - [r = 3]$	
$r_i \leftarrow r_i + r_j \times r_{j'}$	$r_i \leftarrow r_i + r_j \times r_{j'}$	$r_i \leftarrow r_i + r_j \times r_{j'}$	

Running in parallel reduces to 4 recursive calls instead of $4k^2$. The catch: need $3k$ registers instead of 3.



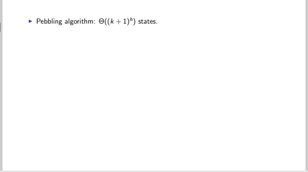
- for $(y, z) \in [k]^2$:
  - $r_i \leftarrow r_i + [f_v(y,z) = x] \times [\ell = y] \times [r = z]$

$r_j \leftarrow r_j + [\ell = 1]$	$r_j \leftarrow r_j + [\ell = 1]$	$r_j \leftarrow r_j + [\ell = 1]$	
$r_i \leftarrow r_i - r_j \times r_{j'}$	$r_i \leftarrow r_i - r_j \times r_{j'}$	$r_i \leftarrow r_i - r_j \times r_{j'}$	
$r_{j'} \leftarrow r_{j'} + [r = 1]$	$r_{j'} \leftarrow r_{j'} + [r = 2]$	$r_{j'} \leftarrow r_{j'} + [r = 3]$	…
$r_i \leftarrow r_i + r_j \times r_{j'}$	$r_i \leftarrow r_i + r_j \times r_{j'}$	$r_i \leftarrow r_i + r_j \times r_{j'}$	…
$r_j \leftarrow r_j - [\ell = 1]$	$r_j \leftarrow r_j - [\ell = 1]$	$r_j \leftarrow r_j - [\ell = 1]$	
$r_i \leftarrow r_i - r_j \times r_{j'}$	$r_i \leftarrow r_i - r_j \times r_{j'}$	$r_i \leftarrow r_i - r_j \times r_{j'}$	…
$r_{j'} \leftarrow r_{j'} - [r = 1]$	$r_{j'} \leftarrow r_{j'} - [r = 2]$	$r_{j'} \leftarrow r_{j'} - [r = 3]$	
$r_i \leftarrow r_i + r_j \times r_{j'}$	$r_i \leftarrow r_i + r_j \times r_{j'}$	$r_i \leftarrow r_i + r_j \times r_{j'}$	

Running in parallel reduces to 4 recursive calls instead of $4k^2$. The catch: need $3k$ registers instead of 3.

This means that instead of four k squared recursive calls, we only need to make four! The catch is that instead of three registers, we need three k, since each recursive call needs to return k different indicator values.
We can think of the output of the subroutine as a k-bit string, where exactly one of the bits is one and the others are zero. We call this a *one-hot encoding*.
So, how efficient is this strategy?

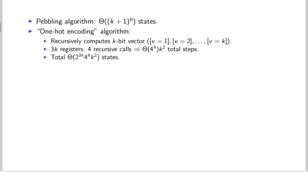
- Pebbling algorithm: $\Theta((k+1)^h)$ states.



- Pebbling algorithm: $\Theta((k+1)^h)$ states.

Remember that the Pebbling algorithm uses on the order of k plus one to the h states.

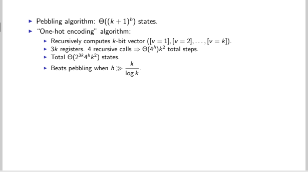
- Pebbling algorithm: $\Theta((k+1)^h)$ states.
- "One-hot encoding" algorithm:
  - Recursively computes k-bit vector $([v=1],[v=2],\ldots,[v=k])$.
  - $3k$ registers. 4 recursive calls $\Rightarrow \Theta(4^h)k^2$ total steps.
  - Total $\Theta(2^{3k}4^hk^2)$ states.



- Pebbling algorithm: $\Theta((k+1)^h)$ states.
- "One-hot encoding" algorithm:
  - Recursively computes k-bit vector $([v=1],[v=2],\ldots,[v=k])$.
  - $3k$ registers. 4 recursive calls $\Rightarrow \Theta(4^h)k^2$ total steps.
  - Total $\Theta(2^{3k}4^hk^2)$ states.

This new parallel algorithm uses three k registers, so each layer of the branching program will have two to the three k states. It calls itself recursively four times, which means the number of layers is on the order of four to the h times k squared extra work that needs to be done.
In total, we have on the order of two to the three k times four to the h times k squared states.

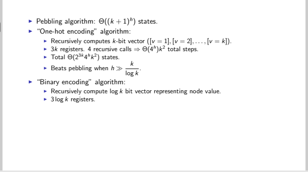
- Pebbling algorithm: $\Theta((k+1)^h)$ states.
- "One-hot encoding" algorithm:
  - Recursively computes k-bit vector $([v=1],[v=2],\ldots,[v=k])$.
  - $3k$ registers. 4 recursive calls $\Rightarrow \Theta(4^h)k^2$ total steps.
  - Total $\Theta(2^{3k}4^h k^2)$ states.
  - Beats pebbling when $h \gg \dfrac{k}{\log k}$.



- Pebbling algorithm: $\Theta((k+1)^h)$ states.
- "One-hot encoding" algorithm:
  - Recursively computes k-bit vector $([v=1],[v=2],\ldots,[v=k])$.
  - $3k$ registers. 4 recursive calls $\Rightarrow \Theta(4^h)k^2$ total steps.
  - Total $\Theta(2^{3k}4^hk^2)$ states.
  - Beats pebbling when $h \gg \frac{k}{\log k}$.

When k is large compared to h, this is much worse than the pebbling algorithm. But when h is asymptotically larger than around k over log k, this algorithm is an improvement.
The three times k registers are really hurting us, so the next thing we tried was a binary encoding: instead of k bits, use log k bits to represent the value at the node in binary.

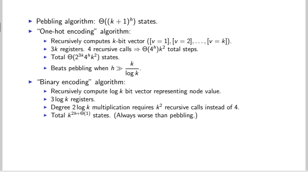
- Pebbling algorithm: $\Theta((k+1)^h)$ states.
- "One-hot encoding" algorithm:
  - Recursively computes k-bit vector $([v=1],[v=2],\ldots,[v=k])$.
  - $3k$ registers. 4 recursive calls $\Rightarrow \Theta(4^h)k^2$ total steps.
  - Total $\Theta(2^{3k}4^h k^2)$ states.
  - Beats pebbling when $h \gg \dfrac{k}{\log k}$.
- "Binary encoding" algorithm:
  - Recursively compute $\log k$ bit vector representing node value.
  - $3\log k$ registers.



- Pebbling algorithm: $\Theta((k+1)^h)$ states.
- "One-hot encoding" algorithm:
  - Recursively computes k-bit vector $([v=1],[v=2],\ldots,[v=k])$.
  - $3k$ registers. 4 recursive calls $\Rightarrow \Theta(4^h)k^2$ total steps.
  - Total $\Theta(2^{3k}4^h k^2)$ states.
  - Beats pebbling when $h \gg \frac{k}{\log k}$.
- "Binary encoding" algorithm:
  - Recursively compute $\log k$ bit vector representing node value.
  - $3\log k$ registers.

The benefit here is that we only need three times log k registers, instead of three k.

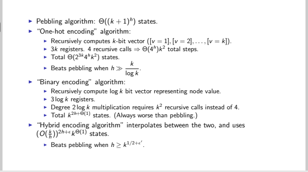
- Pebbling algorithm: $\Theta((k+1)^h)$ states.
- "One-hot encoding" algorithm:
  - Recursively computes k-bit vector $([v=1],[v=2],\ldots,[v=k])$.
  - $3k$ registers. 4 recursive calls $\Rightarrow \Theta(4^h)k^2$ total steps.
  - Total $\Theta(2^{3k}4^h k^2)$ states.
  - Beats pebbling when $h \gg \dfrac{k}{\log k}$.
- "Binary encoding" algorithm:
  - Recursively compute $\log k$ bit vector representing node value.
  - $3\log k$ registers.
  - Degree $2\log k$ multiplication requires k^2 recursive calls instead of 4.
  - Total $k^{2h+\Theta(1)}$ states. (Always worse than pebbling.)



- Pebbling algorithm: $\Theta((k+1)^h)$ states.
- "One-hot encoding" algorithm:
  - Recursively computes k-bit vector $([v=1],[v=2],\dots,[v=k])$.
  - $3k$ registers. 4 recursive calls $\Rightarrow \Theta(4^h)k^2$ total steps.
  - Total $\Theta(2^{3k}4^hk^2)$ states.
  - Beats pebbling when $h \gg \frac{k}{\log k}$.
- "Binary encoding" algorithm:
  - Recursively compute $\log k$ bit vector representing node value.
  - $3\log k$ registers.
  - Degree $2\log k$ multiplication requires k^2 recursive calls instead of 4.
  - Total $k^{2h+\Theta(1)}$ states. (Always worse than pebbling.)

The trouble is that we end up needing to multiply more than two values at once — our formula has degree two log k. We found that we needed to make k squared recursive calls in order to multpily two log k values, resulting in a total of k to the two h plus order one states.
This is strictly worse than the pebbling algorithm, but it's still a useful stepping stone.

- Pebbling algorithm: $\Theta((k+1)^h)$ states.
- "One-hot encoding" algorithm:
  - Recursively computes k-bit vector $([v=1],[v=2],\ldots,[v=k])$.
  - $3k$ registers. 4 recursive calls $\Rightarrow \Theta(4^h)k^2$ total steps.
  - Total $\Theta(2^{3k}4^h k^2)$ states.
  - Beats pebbling when $h \gg \dfrac{k}{\log k}$.
- "Binary encoding" algorithm:
  - Recursively compute $\log k$ bit vector representing node value.
  - $3 \log k$ registers.
  - Degree $2 \log k$ multiplication requires k^2 recursive calls instead of 4.
  - Total $k^{2h+\Theta(1)}$ states. (Always worse than pebbling.)
- "Hybrid encoding algorithm" interpolates between the two, and uses $(O(\frac{k}{h}))^{2h+\epsilon}k^{\Theta(1)}$ states.
  - Beats pebbling when $h \geq k^{1/2+\epsilon'}$.

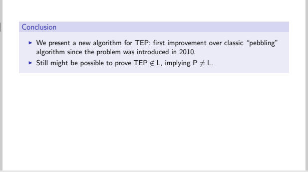


- Pebbling algorithm: $\Theta((k+1)^h)$ states.
- "One-hot encoding" algorithm:
  - Recursively computes k-bit vector $([v=1],[v=2],\ldots,[v=k])$.
  - $3k$ registers. 4 recursive calls $\Rightarrow \Theta(4^h)k^2$ total steps.
  - Total $\Theta(2^{3k}4^hk^2)$ states.
  - Beats pebbling when $h \gg \frac{k}{\log k}$.
- "Binary encoding" algorithm:
  - Recursively compute $\log k$ bit vector representing node value.
  - $3\log k$ registers.
  - Degree $2\log k$ multiplication requires k^2 recursive calls instead of 4.
  - Total $k^{2h+\Theta(1)}$ states. (Always worse than pebbling.)
- "Hybrid encoding algorithm" interpolates between the two, and uses $(O(\frac{k}{h}))^{2h+\epsilon}k^{\Theta(1)}$ states.
  - Beats pebbling when $h \geq k^{1/2+\epsilon'}$.

By interpolating between the two encodings, you get an algorithm that does asymptotically better than pebbling as long as the height of the tree is at least k to the power one half plus any small constant.

Conclusion

- We present a new algorithm for TEP: first improvement over classic “pebbling” algorithm since the problem was introduced in 2010.
- Still might be possible to prove TEP $\notin$ L, implying P $\neq$ L.



Conclusion

- We present a new algorithm for TEP: first improvement over classic “pebbling” algorithm since the problem was introduced in 2010.
- Still might be possible to prove TEP $\notin$ L, implying P $\neq$ L.

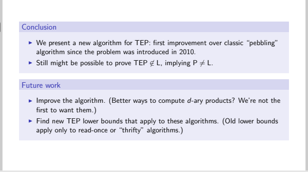
In conclusion, we presented a new algorithm for the tree evaluation problem, which is the first improvement since TEP was introduced ten years ago.
It is not a log space algorithm, so TEP remains a possible approach for separating P from L.

Conclusion

- We present a new algorithm for TEP: first improvement over classic "pebbling" algorithm since the problem was introduced in 2010.
- Still might be possible to prove TEP $\notin$ L, implying P $\neq$ L.

Future work

- Improve the algorithm. (Better ways to compute *d*-ary products? We're not the first to want them.)
- Find new TEP lower bounds that apply to these algorithms. (Old lower bounds apply only to read-once or "thrifty" algorithms.)


Conclusion

- We present a new algorithm for TEP: first improvement over classic "pebbling" algorithm since the problem was introduced in 2010.
- Still might be possible to prove TEP $\notin$ L, implying P $\neq$ L.

Future work

- Improve the algorithm. (Better ways to compute d-ary products? We're not the first to want them.)
- Find new TEP lower bounds that apply to these algorithms. (Old lower bounds apply only to read-once or "thrifty" algorithms.)

There are two basic directions for future work.
The first is to improve the algorithm. The main limiting factor seems to be computing products. The "binary encoding" algorithm didn't work because the number of recursive calls we have to make at each level is exponential in the degree of the polynomial we're computing. It would be nice to be able to improve that. We're not the first to point out this direction.
The other direction is to go back to proving lower bounds for the tree evaluation problem. If you remember, I briefly mentioned that we have lower bounds for two restricted classes of algorithm. The first is read-once algorithms, which are never allowed to read the same part of the input twice. The second is thrifty algorithms, which never read an irrelevant piece of the input. Our new algorithms violate both of those restrictions: we read every single part of the input, whether it's relevent or not, and we do it over and over again, using repeated computation to save memory

Thanks!

Thanks for watching, and I hope to see you at the first fully online STOC!