Borrowing memory that's being used: catalytic approaches to the Tree Evaluation Problem

James Cook, Ian Mertz

April 6, 2020

Borrowing memory that's being used: catalytic approaches to the Tree Evaluation Problem

Borrowing memory that's being used: catalytic approaches to the Tree Evaluation Problem

James Cook, Ian Mertz

April 6, 2020

Today I'm presenting some joint work with Ian Mertz scheduled to appear at STOC 2020. This talk has two parts.

First I'll tell you about the Tree Evaluation Problem and why we care about it. Then I'll show you a new algorithm for solving it, based on an idea around borrowing memory. I'll explain more about that later.

The Tree Evaluation 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]

Section 1

The Tree Evaluation Problem

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.
I'll start by defining the problem after giving some motivation explaining why we care about it. 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 new algorithm had to work around.
So, let's start with the motivation, which is separating complexity classes. I want to start by illustrating an embarassing situation in complexity theory.

$$AC^0(6) \subseteq L \subseteq P \subseteq NP \subseteq PH$$

$AC^0(6) \subseteq L \subseteq P \subseteq NP \subseteq PH$

Here's a sequence of complexity classes. Don't worry if you don't know what all these are. On the right side, we have NP followed by the whole polynomial hierarchy.
On the left, we have uniform AC zero of six. As far as we know, AC zero of six is a really weak complexity class. For example, given a string of bits, we don't know how to count whether most of the bits are zero or one.
And in between these two extremes, there's a wide range of complexity classes that I could add to this slide that are all distinct from each other — as far as we know.

$$\mathsf{AC}^0(6) \subseteq \mathsf{L} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PH}$$

We don't know whether $\mathsf{AC}^0(6) = \mathsf{PH}$.

$AC^0(6) \subseteq L \subseteq P \subseteq NP \subseteq PH$

We don't know whether $AC^0(6) = PH$.

The embarassing thing is that we still don't have any proof that these classes aren't all the same as each other. Any kind of separation on this line would be a breakthrough result.

$$AC^0(6) \subseteq L \subseteq P \subseteq NP \subseteq PH$$

We don't know whether $AC^0(6) = PH$.

$AC^0(6) \subseteq L \subseteq P \subseteq NP \subseteq PH$

We don't know whether $AC^0(6) = PH$.

Today we're going to look at just two of these classes, L and P. The The Tree Evaluation Problem was introduced as an attempt to separate these two.

P = "polynomial time": $O(n^{O(1)})$ time.

P = "polynomial time": $O(n^{O(1)})$ time.

Just to make sure everyone's on the same page, P stands for *polynomial time*. It's the class of decision problems which can be solved by a Turing machine that runs for a number of steps that's polynomial in n, where n is the length of the input, measured in bits.

P = "polynomial time": $O(n^{O(1)})$ time.

L = "logarithmic space": $O(\log n)$ memory.
$2^{O(\log n)} = n^{O(1)}$ configurations, so L $\subseteq$ P.

P = "polynomial time": $O(n^{O(1)})$ time.

L = "logarithmic space": $O(\log n)$ memory.
$2^{O(\log n)} = n^{O(1)}$ configurations, so L $\subseteq$ P.

Our other class is L, which stands for *logarithmic space*. Here, the constraint is that the Turing machine can use big oh of log n memory.
That means the Turing machine can have two to the power big oh of log n different configurations, which is the same thing as a polynomial in n. Since we can never repeat a configuration without looping forever, this tells us that every logspace Turing machine is a polynomial time Turing machine, so L is a subset of P.
The Tree Evaluation Problem was introduced in an effort to prove that this inclusion is strict.

P = "polynomial time": $O(n^{O(1)})$ time.

L = "logarithmic space": $O(\log n)$ memory.
$2^{O(\log n)} = n^{O(1)}$ configurations, so L $\subseteq$ P.

TEP $\in$ P.
Goal: prove TEP $\notin$ L, so L $\neq$ P.

P = "polynomial time": $O(n^{O(1)})$ time.

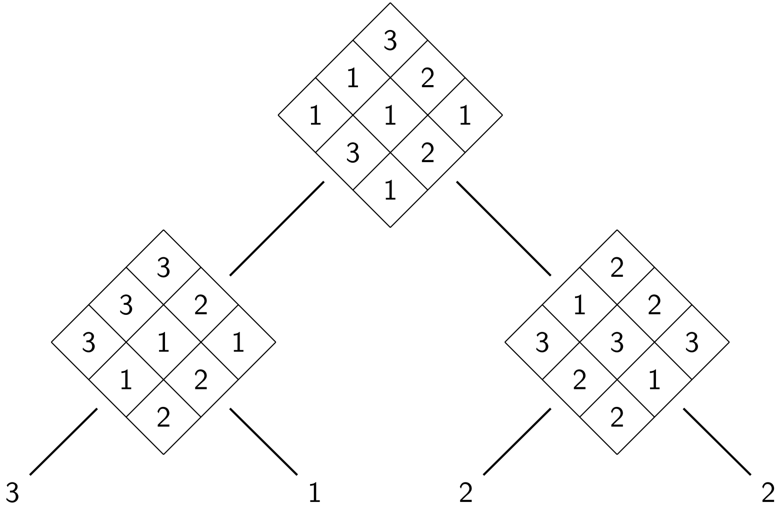
L = "logarithmic space": $O(\log n)$ memory.
$2^{O(\log n)} = n^{O(1)}$ configurations, so $\mathsf{L} \subseteq \mathsf{P}$.

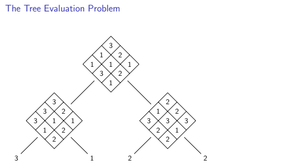
TEP $\in$ P.
Goal: prove TEP $\notin$ L, so L $\neq$ P.

The tree evaluation problem, which I'll write as TEP, is easy to solve in polynomial time. But it seems like it should be impossible to solve in log space. The hope of people who study the Tree Evaluation Problem is to prove that it is indeed not in L, which would imply that L is not equal to P.

So, that's the motivation. Now let's talk about what this problem actually is.

The Tree Evaluation Problem





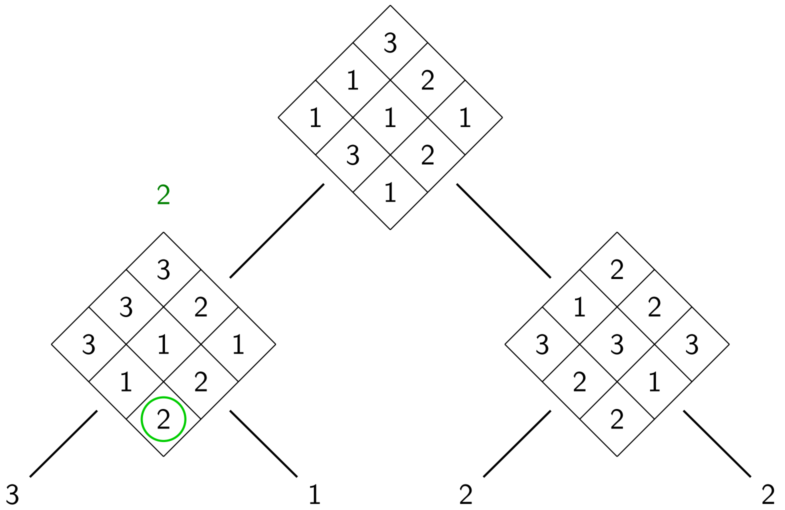
The input to the Tree Evaluation Problem is a complete binary tree with some information attached to each node. Each leaf has a number attached to it — in this case, 3, 1, 2 and 2 — and each internal node has a table of numbers.

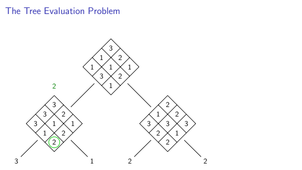
Given that input, we're going to recursively define a single number at each node, called the value of the node.

The 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. Those two values tell us where to look in that internal node's table. For the internal node on the left, we look at the entry in row three, column one of its table, and we find the number two.

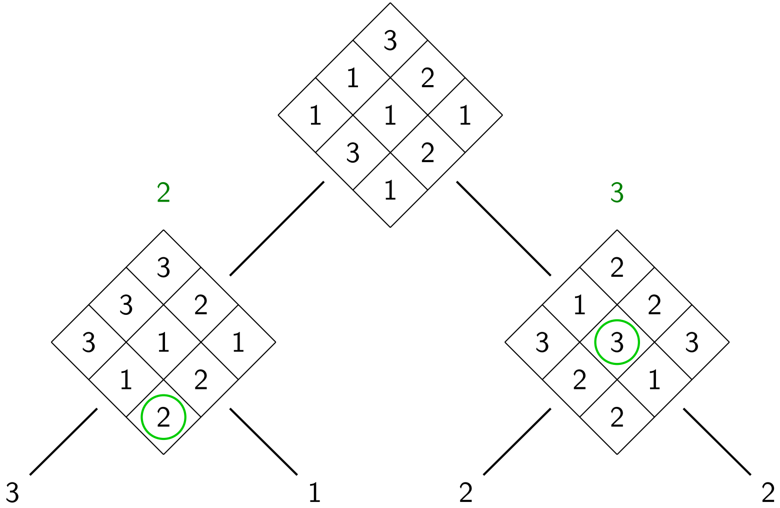
The Tree Evaluation Problem



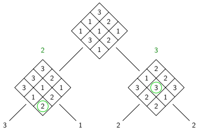


Similarly, we look up row two column two of the node on the right, and find the number three.

The Tree Evaluation Problem

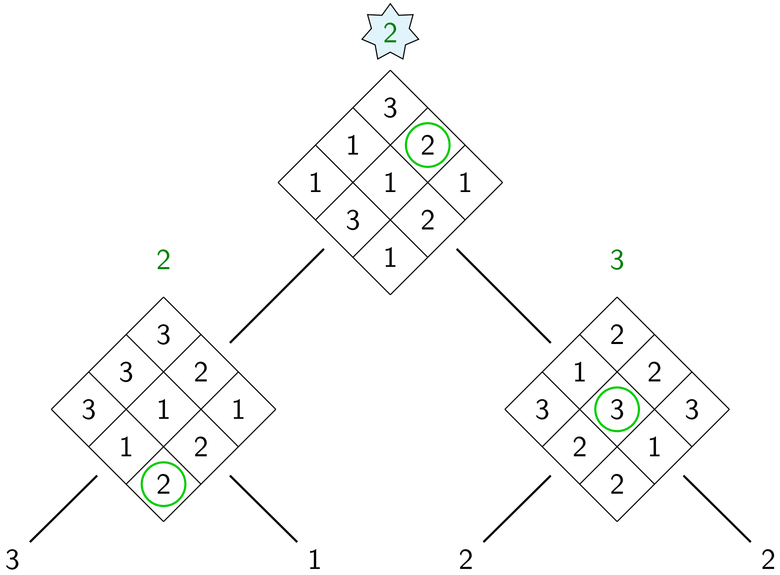


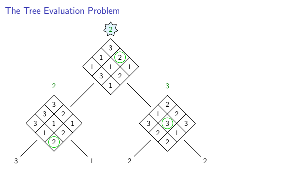
The Tree Evaluation Problem



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

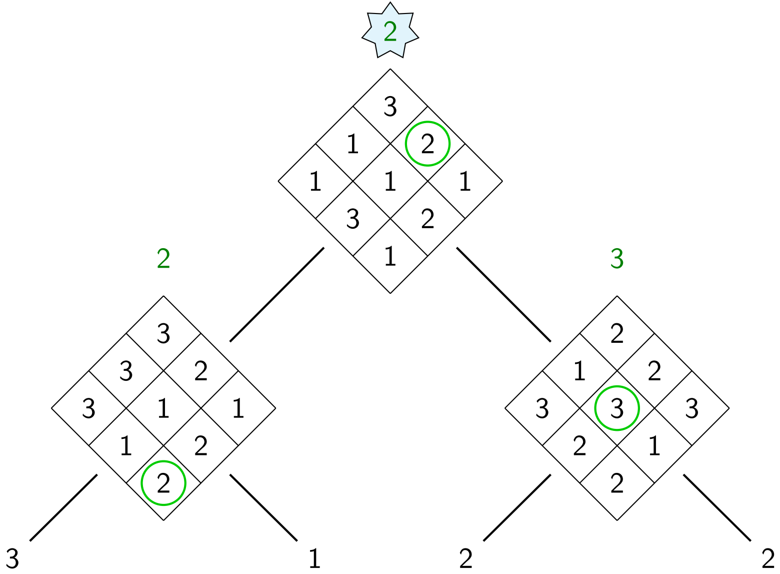
The Tree Evaluation Problem





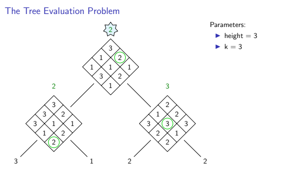
The output of the Tree Evaluation Problem is the value at the root. To turn this into a decision problem, we can say the output is true iff the value at the root is one.

The Tree Evaluation Problem



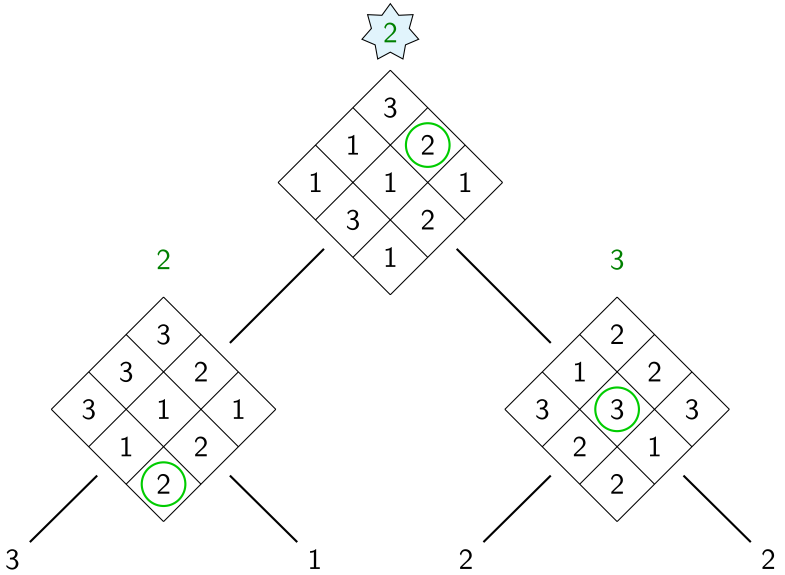
Parameters:

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



There are two parameters to this problem. The first parameter 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.

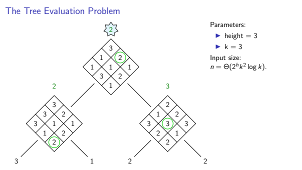
The Tree Evaluation Problem



Parameters:

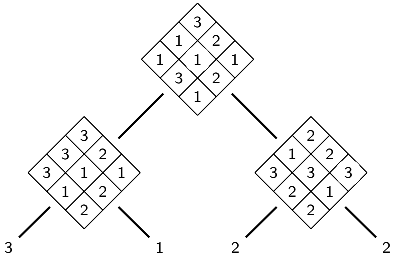
- height = 3
- k = 3

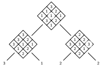
Input size:
$n = \Theta(2^h k^2 \log k)$.



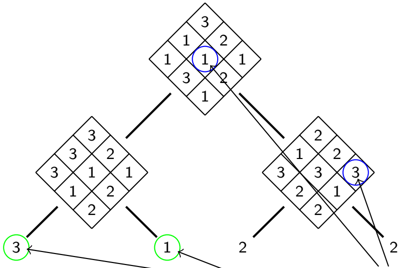
In general, the size of the input is on the order of two to the h internal nodes, times k squared numbers stored in each node, times log k bits to store each number.

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.

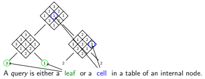

3
1 2
1 1 1
3 2
1
3
3 2
3 1 1
1 2
2
2
1 2
3 3 3
2 1
2
3
1
2
2



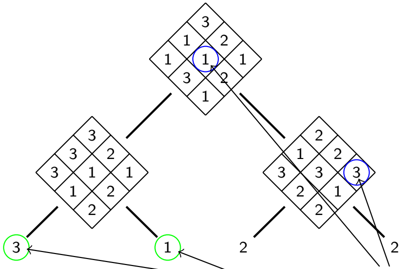
So, here's our TEP input again. I'll define a *query* to be any piece of that input we might want to read.



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



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 branching program is a directed graph, where the nodes are called states. Each state is labelled by a query, and each edge is labelled by the answer to a query.

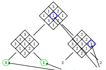


A *query* is either a leaf or a cell in a table of an internal node.

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.



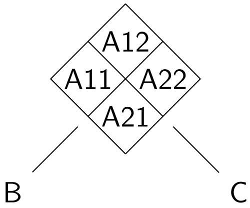
A *query* is either a leaf or a cell in a table of an internal node.

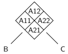
A *branching program* is a directed graph of *states*. There are two kinds of state:

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

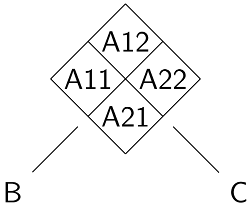
One state is the starting state.

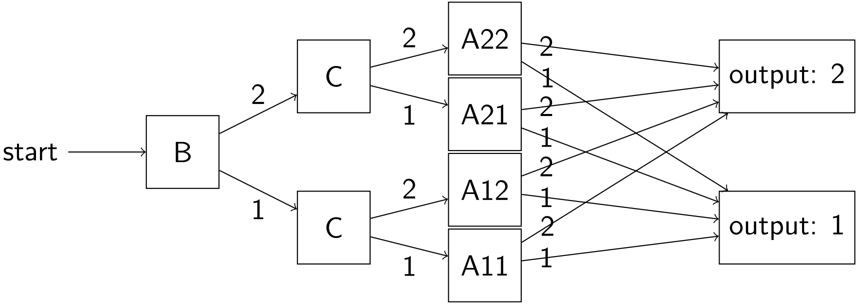
To be more precise, there are two kinds of state. A query state is labelled with a query, and has k outgoing edges: the edge you follow depends on the answer to the query. The other kind is a final state. When you get to one of those, the computation stops, and you output whatever the state is labelled with. And one of the states is marked as the starting state.
As an example, let's see a branching program that solves the tree evaluation problem when the height and alphabet size k are both two.

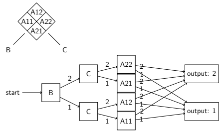

A12
A11
A22
A21
B
C



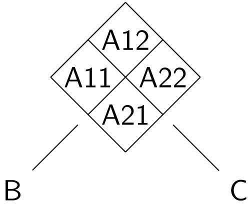
Here's what a height two instance looks like when the alphabet size k equals 2. There are six things we can query: the four cells in the root node A's table, and the two leaves B and C.

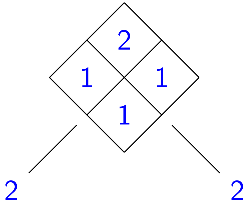

A12
A11
A22
A21
B
C

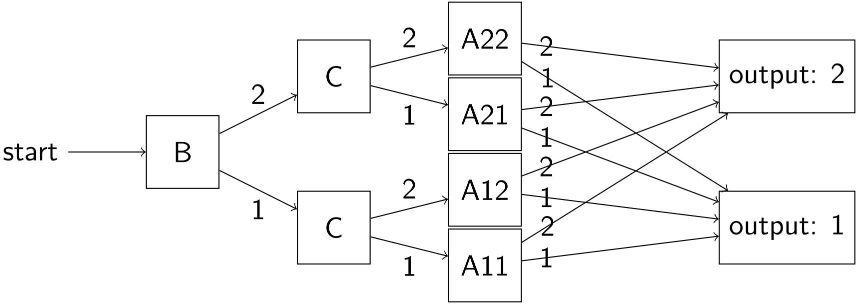

start
B
2
1
C
C
2
1
2
1
A22
A21
A12
A11
2
1
2
1
2
1
2
1
output: 2
output: 1

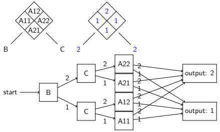


Here's a branching program that solves it. It's organized into *layers* going from left to right. The starting state queries the first leaf, B. Depending on the answer, we end up in one of the two states in the next layer. Those states query the other leaf C, and depending on the answer, we end up in 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.

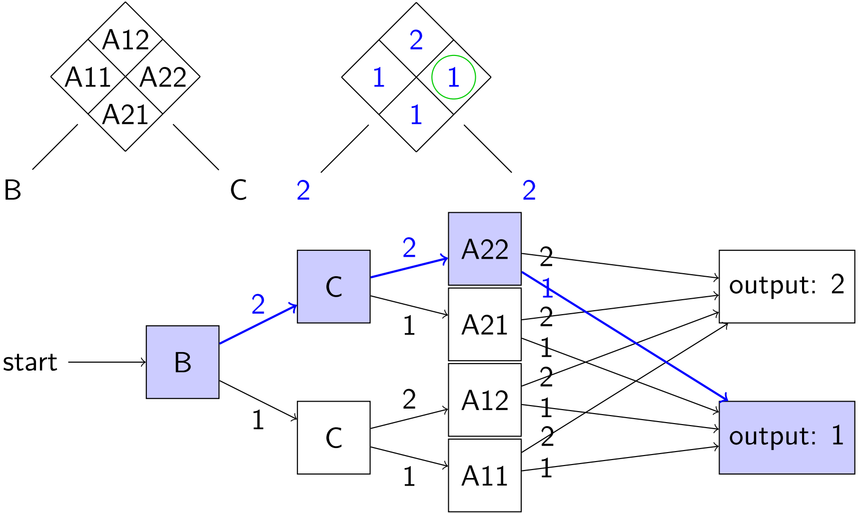

A12
A11
A22
A21
B
C

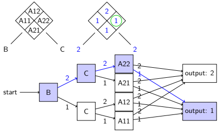

2
1
1
1
2
2


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

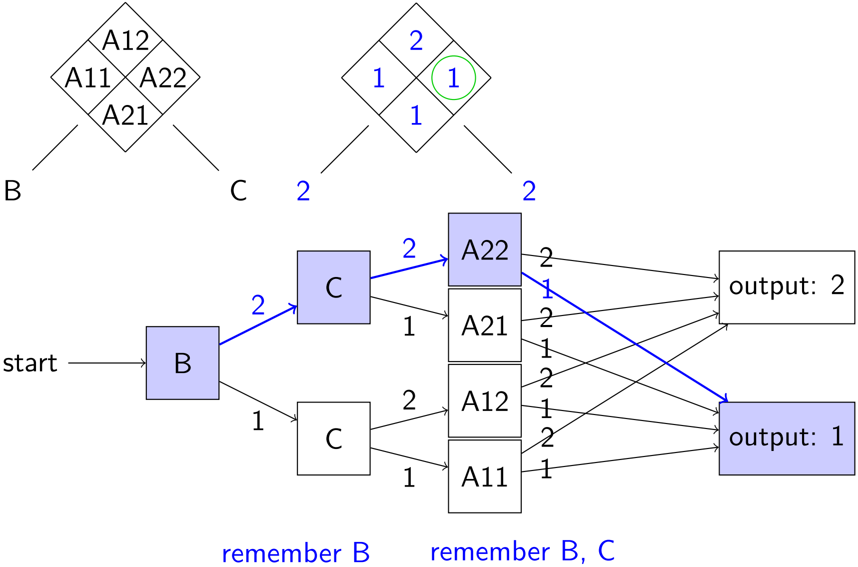


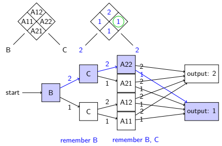
Here's an example input. Let's see what the computation looks like.


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


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



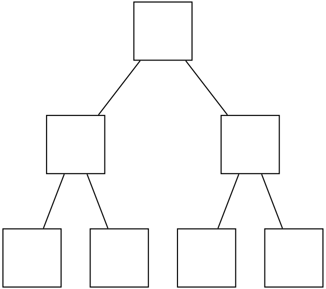
In the second layer, we remember node B, and in the third layer, we remember both B and C. The lower bounds we have so far all involve arguments about how many things the branching program needs to remember at once.

One way to model this idea of remembering things is pebbling games.

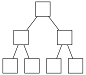
Pebbling game [Paterson Hewitt 1970]

Pebbling 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]

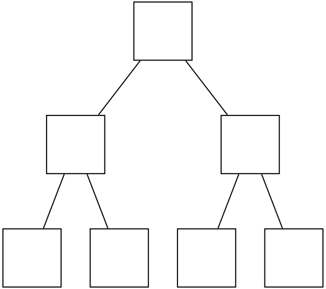


Pebbling game [Paterson Hewitt 1970]



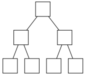
Three in this case.

Pebbling game [Paterson Hewitt 1970]



Limited supply of pebbles (say, 3).

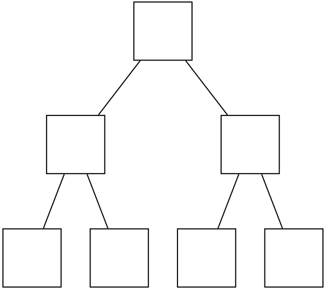
Pebbling game [Paterson Hewitt 1970]



Limited supply of pebbles (say, 3).

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

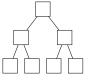
Pebbling game [Paterson Hewitt 1970]



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]

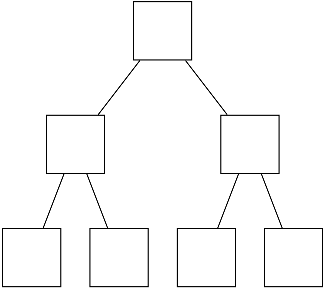


Limited supply of pebbles (say, 3).
Two kinds of move:

- 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]

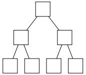


Limited supply of pebbles (say, 3).
Two kinds of move:

- 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]



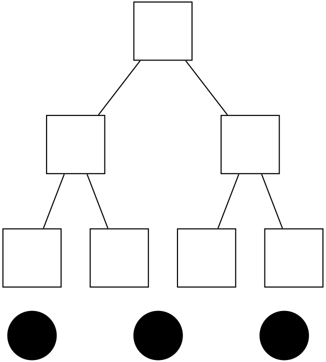
Limited supply of pebbles (say, 3).
Two kinds of move:

- 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.

Let's try.

Pebbling game [Paterson Hewitt 1970]

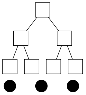


Limited supply of pebbles (say, 3).
Two kinds of move:

- 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]



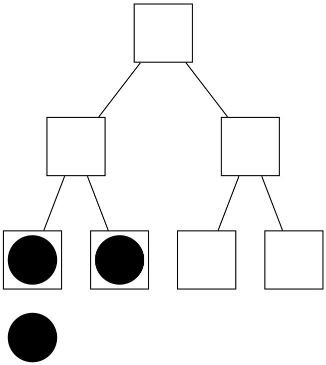
Limited supply of pebbles (say, 3).
Two kinds of move:

- 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 have three pebbles. We'll start by moving two of the pebbles to the leftmost two leaves.

Pebbling game [Paterson Hewitt 1970]

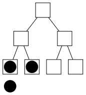


Limited supply of pebbles (say, 3).
Two kinds of move:

- 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]



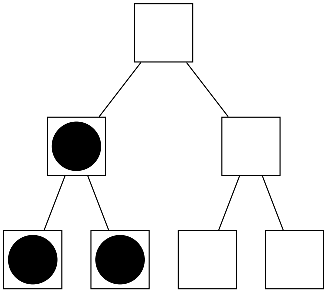
Limited supply of pebbles (say, 3).
Two kinds of move:

- 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.

Now, the internal node on the left has pebbles on both of its children. So we're allowed to move a pebble to it.

Pebbling game [Paterson Hewitt 1970]

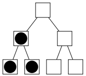


Limited supply of pebbles (say, 3).
Two kinds of move:

- 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]



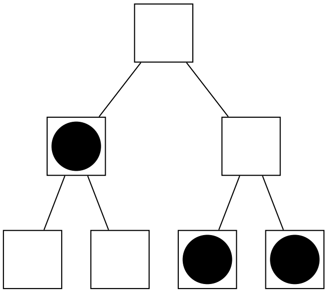
Limited supply of pebbles (say, 3).
Two kinds of move:

- 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.

This is good progress. Our goal is to put a pebble on the root, and we've already got one of the root's two children. Now, let's focus on the right side of the tree. I'll move two pebbles to the two leaves on the right.

Pebbling game [Paterson Hewitt 1970]

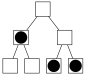


Limited supply of pebbles (say, 3).
Two kinds of move:

- 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]



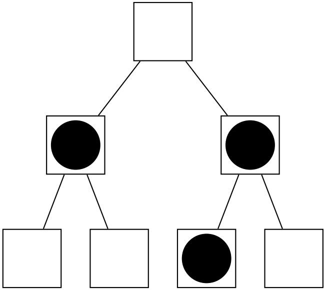
Limited supply of pebbles (say, 3).
Two kinds of move:

- 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.

Now I'm allowed to move a pebble to the right child of the root. Which pebble should I move? Not the one that's already on the root's left child, because I don't want to lose that progress I've made! So, I move one of the other two pebbles to that node.

Pebbling game [Paterson Hewitt 1970]

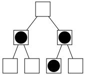


Limited supply of pebbles (say, 3).
Two kinds of move:

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

Goal: put a pebble on the root.

Pebbling game [Paterson Hewitt 1970]



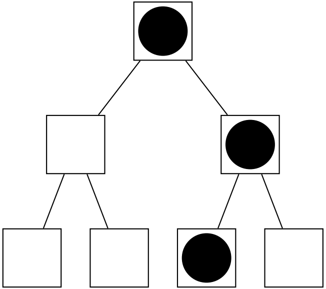
Limited supply of pebbles (say, 3).
Two kinds of move:

- 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.

Now, both of the root's children have pebbles on them, so I can move a pebble to the root.

Pebbling game [Paterson Hewitt 1970]

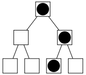


Limited supply of pebbles (say, 3).
Two kinds of move:

- 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]



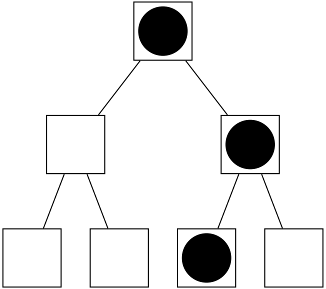
Limited supply of pebbles (say, 3).
Two kinds of move:

- 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.

And we're done. 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]



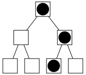
Limited supply of pebbles (say, 3).
Two kinds of move:

- 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:

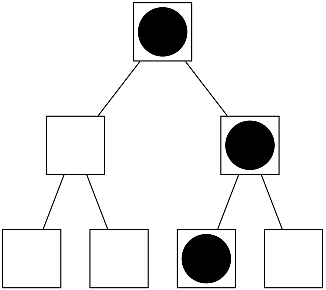
- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

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]



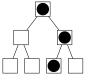
Limited supply of pebbles (say, 3).
Two kinds of move:

- 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:

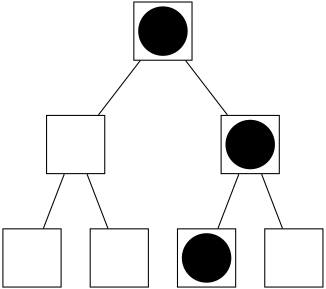
- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

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. The way this works is that each step of the game translates into a layer of our 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]



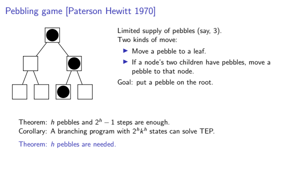
Limited supply of pebbles (say, 3).
Two kinds of move:

- 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.

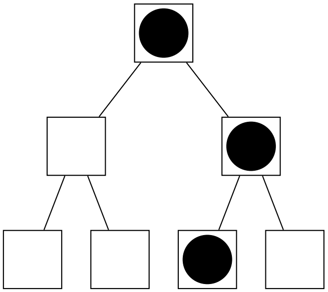
Theorem: h pebbles and $2^h - 1$ steps are enough.
Corollary: A branching program with $2^h k^h$ states can solve TEP.

Theorem: h pebbles are needed.



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:

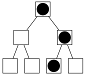
- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

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.

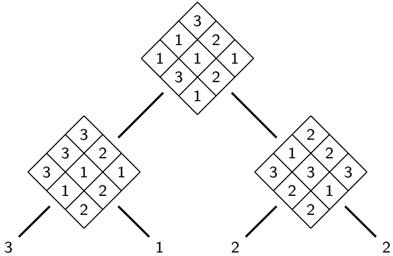
Theorem: h pebbles and $2^h - 1$ steps are enough.
Corollary: A branching program with $2^h k^h$ states can solve TEP.

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

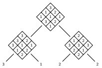
Since we need at least h pebbles, maybe we can prove that the tree evaluation problem needs at least on the order of k to the h states.
For a long time, nobody could come up with any algorithm that did better, so this conjecture seemed quite plausible.
The algorithm I'll present later is the first counterexample.
Now, let's take a moment to see where we are.

We've defined the tree evaluation problem and talked about branching programs and pebbling games for solving it. Now I promised I'd talk about some lower bounds.
If you remember, the goal behind the tree evaluation problem is to separate log space from polynomial time. So, let's see what it would take to prove that this problem can't be solved in log space.
To define what log space means here, we need to measure the size of the input.

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

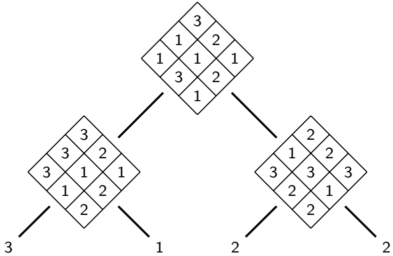


Input size: $\Theta(2^h k^2 \log k)$.

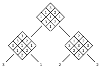


The input consists of a k by k table at each internal node, and a single number at each leaf. So, the size of the input is on the order of two to the h internal nodes times k squared numbers at each node times log k bits to store each number.

Input size: $\Theta(2^h k^2 \log k)$. So, log space $= O(h + \log k)$ memory.



Input size: $\Theta(2^h k^2 \log k)$. So, log space $= O(h + \log k)$ memory.



Taking the logarithm of that means that a log space Turing machine is allowed to use on the order of h plus log k memory.

Now, we want to understand that in terms of branching programs. For a fixed input size, any Turing machine can be transformed into a branching program, with one node for every possible configuration of the Turing machine.

Input size: $\Theta(2^h k^2 \log k)$. So, log space $= O(h + \log k)$ memory.

If TEP $\in$ L, then it can be solved by a family of branching programs with $2^{O(h+\log k)} = 2^{O(h)} k^{O(1)}$ states.

Input size: $\Theta(2^h k^2 \log k)$. So, log space $= O(h + \log k)$ memory.

If TEP $\in$ L, then it can be solved by a family of branching programs with $2^{O(h+\log k)} = 2^{O(h)} k^{O(1)}$ states.

That means that if TEP is in log space, then it can be solved by a family of branching programs with two to the order of h plus log k states, which equals two to the order h times k to some constant. I say "family" here because we need a different branching program for each input size. So, we can restate the goal of the Tree Evaluation Problem work as: prove that it cannot be solved by branching programs with two to the order h times k to a constant states. That would imply it's not in L, and so L is not equal to P.

Input size: $\Theta(2^h k^2 \log k)$. So, log space $= O(h + \log k)$ memory.

If TEP $\in$ L, then it can be solved by a family of branching programs with $2^{O(h+\log k)} = 2^{O(h)} k^{O(1)}$ states.

Pebbling algorithm (previous best):

- 2^h layers.
- Up to k^h states per layer.
- Total $\Theta((k+1)^h)$ states.

Input size: $\Theta(2^h k^2 \log k)$. So, log space $= O(h + \log k)$ memory.

If TEP $\in$ L, then it can be solved by a family of branching programs with $2^{O(h+\log k)} = 2^{O(h)} k^{O(1)}$ states.

Pebbling algorithm (previous best):

- 2^h layers.
- Up to k^h states per layer.
- Total $\Theta((k+1)^h)$ states.

Now, the pebbling-based algorithm for TEP, which until now was the best known, has two to the h layers with a varying number of states per layer ranging up to k to the h. If you add up all the layers, it ends up working out to big theta of k plus one to the h states.

Since k is raised to more than a constant power, it is not a log space algorithm.

That false conjecture from the previous slide said that k to the h was a lower bound for all branching programs. If that had turned out to be true, it would have meant that you can't solve TEP in log space.

Input size: $\Theta(2^h k^2 \log k)$. So, log space $= O(h + \log k)$ memory.

If TEP $\in$ L, then it can be solved by a family of branching programs with $2^{O(h+\log k)} = 2^{O(h)} k^{O(1)}$ states.

Pebbling algorithm (previous best):

- 2^h layers.
- Up to k^h states per layer.
- Total $\Theta((k+1)^h)$ states.

New algorithm: $(\frac{k}{h} + 1)^{\Theta(h)} k^{\Theta(1)}$ states. (Beats pebbling when $h \geq k^{4/5+\epsilon}$.)

Input size: $\Theta(2^h k^2 \log k)$. So, log space $= O(h + \log k)$ memory.

If TEP $\in$ L, then it can be solved by a family of branching programs with $2^{O(h+\log k)} = 2^{O(h)} k^{O(1)}$ states.

Pebbling algorithm (previous best):

- 2^h layers.
- Up to k^h states per layer.
- Total $\Theta((k+1)^h)$ states.

New algorithm: $(\frac{k}{h}+1)^{\Theta(h)} k^{\Theta(1)}$ states. (Beats pebbling when $h \geq k^{4/5+\epsilon}$.)

The new algorithm I'm going to show you has k over h plus one to the theta of h times a polynomial in k states.
If you compare it to pebbling, it's better as long as h is not too small compared to k. Specifically, if h is k to a power bigger than four over five, this algorithm is an asymptotic improvement.
But, it's still not log space.

Input size: $\Theta(2^h k^2 \log k)$. So, log space $= O(h + \log k)$ memory.

If TEP $\in$ L, then it can be solved by a family of branching programs with $2^{O(h+\log k)} = 2^{O(h)} k^{O(1)}$ states.

Pebbling algorithm (previous best):

- 2^h layers.
- Up to k^h states per layer.
- Total $\Theta((k+1)^h)$ states.

New algorithm: $(\frac{k}{h}+1)^{\Theta(h)} k^{\Theta(1)}$ states. (Beats pebbling when $h \geq k^{4/5+\epsilon}$.)

Neither algorithm fits in $2^{O(h)} k^{O(1)}$ states, so TEP $\notin$ L is still possible.

Input size: $\Theta(2^h k^2 \log k)$. So, log space $= O(h + \log k)$ memory.

If TEP $\in$ L, then it can be solved by a family of branching programs with $2^{O(h+\log k)} = 2^{O(h)} k^{O(1)}$ states.

Pebbling algorithm (previous best):

- 2^h layers.
- Up to k^h states per layer.
- Total $\Theta((k+1)^h)$ states.

New algorithm: $(\frac{k}{h} + 1)^{\Theta(h)} k^{\Theta(1)}$ states. (Beats pebbling when $h \geq k^{4/5+\epsilon}$.)

Neither algorithm fits in $2^{O(h)} k^{O(1)}$ states, so TEP $\notin$ L is still possible.

Both of these algorithms use more than two to the order h times k to a constant states. So, even though we've ruled out the conjecture that pebbling is the best possible, the door is still open to proving TEP is not solvable in log space.
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. . .

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

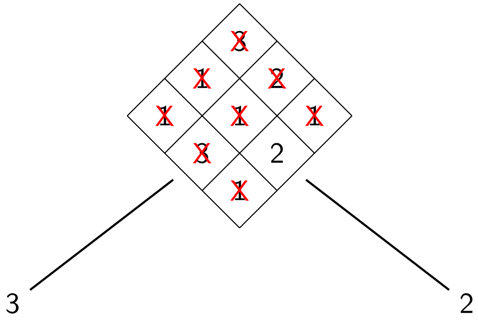
Solving TEP requires $\Omega(k^h)$ states (like the pebbling algorithm) if you assume...

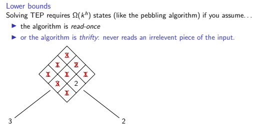
- the algorithm is *read-once*

Lower bounds

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

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





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.
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.

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.

New algorithm: $(\frac{k}{h}+1)^{\Theta(h)}k^{\Theta(1)} \notin \Omega(k^h)$.

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.

New algorithm: $(\frac{k}{h}+1)^{\Theta(h)} k^{\Theta(1)} \notin \Omega(k^h)$.

The new algorithm beats this lower bound, so you can infer that it's not read-once or thrifty. The algorithm is actually going to read every piece of the input several times.

Okay, I've said a lot of mysterious things about the new algorithm, so maybe it's time I told you how it works.

I'll start with some techniques we use related to reversible computation, and then I'll tell you how we apply them to solve TEP.
I'll start with a paper that caught our attention, and showed us we should be looking at reversible computation.

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 once you're done with your computation, you need to return it back the way it was. Imagine your friend has leant you a hard disk that you can use, but when you're finished, you need to give it back with the same data it started with.
I think a natural first thought here is that this extra memory can't possibly help. For example, if you overwrite any data that was stored in it, you'd better keep a copy of that data somewhere else so that you can put it back once you're finished. So it's hard to imagine how you could get any net gain from it. Surprisingly, the result in the paper is that the extra memory does seem 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, ...

It turns out that with only a logarithmic amount of ordinary memory but a polynomial amount of borrowed memory, you can compute uniform T C one circuits. For example, you can compute the determinant of a matrix in T C one, and we don't know how to do with logarithmic memory. We stumbled on this result when we were trying to prove a lower bound for the Tree Evaluation Problem.

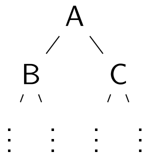
Catalytic space

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

Given:

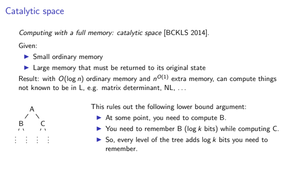
- Small ordinary memory
- Large memory that must be returned to its original state

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.



We had the following idea for a proof. First, at some point you need to compute the left child of the root, B. 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.

Actually, 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.

By the way, how many people are familiar with these results?

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 we're going to use 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 $x_1, \ldots, x_n \in R$
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 $j \neq i$)

We'll say 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 plus f, and every other register is unchanged. Note that the instructions are allowed to use these other registers, as long as they later get restored to their original values.

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.

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 $j \neq i$)

Lemma

Suppose there is a sequence of ℓ *instructions that cleanly computes* f*, and each instruction has the form:*

$$(r_1, \ldots, r_m) \leftarrow g(x_j, r_1, \ldots, r_m)$$

Then there is a branching program that computes f *with* $\ell |R|^m$ *states.*

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 $j \neq i$)

Lemma

Suppose there is a sequence of ℓ instructions that cleanly computes f, and each instruction has the form:

$$(r_1, \ldots, r_m) \leftarrow g(x_j, r_1, \ldots, r_m)$$

Then there is a branching program that computes f with $\ell|R|^m$ states.

In particular, suppose we have a sequence of ℓ instructions that cleanly computes some function f. The instructions are allowed to have a pretty general form. In fact, we'll say that an instruction can update the state of all the registers according to any function g. The only restriction is that g is only allowed to depend on one of the inputs. This restriction comes from the fact that each state of a branching program can only query a single piece of the input.

Then we can convert that into a branching program that uses ℓ times the size of the ring to the power m states. The way we do that is that we build one layer for each instruction in the program. Each layer has R to the m states, so that it can remember all of the register values. The states within that layer all query input x j, and the edges to the next layer are determined according to the function g, since at the current node, you know the values of all the other registers.

So, our overall plan is this. We're going to build an algorithm that solves the Tree Evaluation Problem using a clean computation. Then we'll use this lemma to convert it to a branching program.

Now, let's try some examples of clean computation.

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 $j \neq i$)

Example

Cleanly compute $x_1 + x_2$ into r_1:

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

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 $j \neq i$)

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.

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 $j \neq i$)

Example

Cleanly compute $x_1 + x_2$ into r_1:

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

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 $j \neq i$)

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.

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 $j \neq i$)

Example

Cleanly compute $x_1 + x_2$ into r_1:

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

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 $j \neq i$)

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.

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:

- $r_3 \leftarrow r_3 + r_1 \times r_2$
- 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}

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:

- $r_3 \leftarrow r_3 + r_1 \times r_2$
- 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}

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:

- $r_3 \leftarrow r_3 + r_1 \times r_2$
- 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}

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:

- $r_3 \leftarrow r_3 + r_1 \times r_2$
- 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}

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:

- $r_3 \leftarrow r_3 + r_1 \times r_2$
- 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}

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:

- $r_3 \leftarrow r_3 + r_1 \times r_2$
- 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}

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:

- $r_3 \leftarrow r_3 + r_1 \times r_2$ $[r_3 = \tau_3 + \tau_1 \times \tau_2]$
- 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}

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:

- $r_3 \leftarrow r_3 + r_1 \times r_2$ $[r_3 = \tau_3 + \tau_1 \times \tau_2]$
- 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}

The effect of the first step is straightforward: r3 is now equal to its original value tau3 plus tau1 times tau2.

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:

- $r_3 \leftarrow r_3 + r_1 \times r_2$ $\quad [r_3 = \tau_3 + \tau_1 \times \tau_2]$
- P_1 $\quad [r_1 = \tau_1 + f_1, r_2 = \tau_2]$
- $r_3 \leftarrow r_3 - r_1 \times r_2$ $\quad [r_3 = \tau_3 - f_1 \times \tau_2]$
- 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}

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:

- $r_3 \leftarrow r_3 + r_1 \times r_2$ $[r_3 = \tau_3 + \tau_1 \times \tau_2]$
- P_1 $[r_1 = \tau_1 + f_1, r_2 = \tau_2]$
- $r_3 \leftarrow r_3 - r_1 \times r_2$ $[r_3 = \tau_3 - f_1 \times \tau_2]$
- 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}

After we run P1, register 1 is set to f1 plus its original value tau 1. So, the next instruction takes away the tau1 times tau2 we added in the last step, but also subtracts f times tau two.

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:

- $r_3 \leftarrow r_3 + r_1 \times r_2$ $\quad [r_3 = \tau_3 + \tau_1 \times \tau_2]$
- P_1 $\quad [r_1 = \tau_1 + f_1, r_2 = \tau_2]$
- $r_3 \leftarrow r_3 - r_1 \times r_2$ $\quad [r_3 = \tau_3 - f_1 \times \tau_2]$
- P_2 $\quad [r_1 = \tau_1 + f_1, r_2 = \tau_2 + f_2]$
- $r_3 \leftarrow r_3 + r_1 \times r_2$ $\quad [r_3 = \tau_3 + \tau_1 \times \tau_2 + \tau_1 \times f_2 + f_1 \times f_2]$
- P_1^{-1}
- $r_3 \leftarrow r_3 - r_1 \times r_2$
- P_2^{-1}

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:

- $r_3 \leftarrow r_3 + r_1 \times r_2$ $[r_3 = \tau_3 + \tau_1 \times \tau_2]$
- P_1 $[r_1 = \tau_1 + f_1, r_2 = \tau_2]$
- $r_3 \leftarrow r_3 - r_1 \times r_2$ $[r_3 = \tau_3 - f_1 \times \tau_2]$
- P_2 $[r_1 = \tau_1 + f_1, r_2 = \tau_2 + f_2]$
- $r_3 \leftarrow r_3 + r_1 \times r_2$ $[r_3 = \tau_3 + \tau_1 \times \tau_2 + \tau_1 \times f_2 + f_1 \times f_2]$
- P_1^{-1}
- $r_3 \leftarrow r_3 - r_1 \times r_2$
- P_2^{-1}

Next, we run p two, so register two now has its original value plus f2.
When we add r1 times r2 to r3, we cancel out the f1 times tau2 from the last step. Instead, we add back tau1 times tau2, and also add tau1 times f2 and f1 times f1.
f1 times f2 is our goal. So, all we have to do now is get rid of the terms we don't want, like tau1 times tau2 and tau1 times f2.

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:

- $r_3 \leftarrow r_3 + r_1 \times r_2$ $\quad [r_3 = \tau_3 + \tau_1 \times \tau_2]$
- P_1 $\quad [r_1 = \tau_1 + f_1, r_2 = \tau_2]$
- $r_3 \leftarrow r_3 - r_1 \times r_2$ $\quad [r_3 = \tau_3 - f_1 \times \tau_2]$
- P_2 $\quad [r_1 = \tau_1 + f_1, r_2 = \tau_2 + f_2]$
- $r_3 \leftarrow r_3 + r_1 \times r_2$ $\quad [r_3 = \tau_3 + \tau_1 \times \tau_2 + \tau_1 \times f_2 + f_1 \times f_2]$
- P_1^{-1} $\quad [r_1 = \tau_1, r_2 = \tau_2 + f_2]$
- $r_3 \leftarrow r_3 - r_1 \times r_2$ $\quad [r_3 = \tau_3 + f_1 \times f_2]$
- P_2^{-1}

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:

- $r_3 \leftarrow r_3 + r_1 \times r_2 \qquad [r_3 = \tau_3 + \tau_1 \times \tau_2]$
- $P_1 \qquad [r_1 = \tau_1 + f_1, r_2 = \tau_2]$
- $r_3 \leftarrow r_3 - r_1 \times r_2 \qquad [r_3 = \tau_3 - f_1 \times \tau_2]$
- $P_2 \qquad [r_1 = \tau_1 + f_1, r_2 = \tau_2 + f_2]$
- $r_3 \leftarrow r_3 + r_1 \times r_2 \qquad [r_3 = \tau_3 + \tau_1 \times \tau_2 + \tau_1 \times f_2 + f_1 \times f_2]$
- $P_1^{-1} \qquad [r_1 = \tau_1, r_2 = \tau_2 + f_2]$
- $r_3 \leftarrow r_3 - r_1 \times r_2 \qquad [r_3 = \tau_3 + f_1 \times f_2]$
- P_2^{-1}

We restore register one to its original value by running P1 backward. Then when we subtract r1 times r2, which neatly removes the tau1 times tau1 and tau1 times f2 terms from register three.

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:

- $r_3 \leftarrow r_3 + r_1 \times r_2$ $\quad [r_3 = \tau_3 + \tau_1 \times \tau_2]$
- P_1 $\quad [r_1 = \tau_1 + f_1, r_2 = \tau_2]$
- $r_3 \leftarrow r_3 - r_1 \times r_2$ $\quad [r_3 = \tau_3 - f_1 \times \tau_2]$
- P_2 $\quad [r_1 = \tau_1 + f_1, r_2 = \tau_2 + f_2]$
- $r_3 \leftarrow r_3 + r_1 \times r_2$ $\quad [r_3 = \tau_3 + \tau_1 \times \tau_2 + \tau_1 \times f_2 + f_1 \times f_2]$
- P_1^{-1} $\quad [r_1 = \tau_1, r_2 = \tau_2 + f_2]$
- $r_3 \leftarrow r_3 - r_1 \times r_2$ $\quad [r_3 = \tau_3 + f_1 \times f_2]$
- P_2^{-1} $\quad [r_1 = \tau_1, r_2 = \tau_2]$

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:

- $r_3 \leftarrow r_3 + r_1 \times r_2$ $[r_3 = \tau_3 + \tau_1 \times \tau_2]$
- P_1 $[r_1 = \tau_1 + f_1, r_2 = \tau_2]$
- $r_3 \leftarrow r_3 - r_1 \times r_2$ $[r_3 = \tau_3 - f_1 \times \tau_2]$
- P_2 $[r_1 = \tau_1 + f_1, r_2 = \tau_2 + f_2]$
- $r_3 \leftarrow r_3 + r_1 \times r_2$ $[r_3 = \tau_3 + \tau_1 \times \tau_2 + \tau_1 \times f_2 + f_1 \times f_2]$
- P_1^{-1} $[r_1 = \tau_1, r_2 = \tau_2 + f_2]$
- $r_3 \leftarrow r_3 - r_1 \times r_2$ $[r_3 = \tau_3 + f_1 \times f_2]$
- P_2^{-1} $[r_1 = \tau_1, r_2 = \tau_2]$

All that's left to do is to run P2 backward, so that registers 1 and 2 are restored to their original values.

Register three now holds its original value plus f1 times f2, and the other registers have been restored. That means we're done.

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:

- $r_3 \leftarrow r_3 + r_1 \times r_2 \qquad [r_3 = \tau_3 + \tau_1 \times \tau_2]$
- $P_1 \qquad [r_1 = \tau_1 + f_1, r_2 = \tau_2]$
- $r_3 \leftarrow r_3 - r_1 \times r_2 \qquad [r_3 = \tau_3 - f_1 \times \tau_2]$
- $P_2 \qquad [r_1 = \tau_1 + f_1, r_2 = \tau_2 + f_2]$
- $r_3 \leftarrow r_3 + r_1 \times r_2 \qquad [r_3 = \tau_3 + \tau_1 \times \tau_2 + \tau_1 \times f_2 + f_1 \times f_2]$
- $P_1^{-1} \qquad [r_1 = \tau_1, r_2 = \tau_2 + f_2]$
- $r_3 \leftarrow r_3 - r_1 \times r_2 \qquad [r_3 = \tau_3 + f_1 \times f_2]$
- $P_2^{-1} \qquad [r_1 = \tau_1, r_2 = \tau_2]$

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:

- $r_3 \leftarrow r_3 + r_1 \times r_2$ $[r_3 = \tau_3 + \tau_1 \times \tau_2]$
- P_1 $[r_1 = \tau_1 + f_1, r_2 = \tau_2]$
- $r_3 \leftarrow r_3 - r_1 \times r_2$ $[r_3 = \tau_3 - f_1 \times \tau_2]$
- P_2 $[r_1 = \tau_1 + f_1, r_2 = \tau_2 + f_2]$
- $r_3 \leftarrow r_3 + r_1 \times r_2$ $[r_3 = \tau_3 + \tau_1 \times \tau_2 + \tau_1 \times f_2 + f_1 \times f_2]$
- P_1^{-1} $[r_1 = \tau_1, r_2 = \tau_2 + f_2]$
- $r_3 \leftarrow r_3 - r_1 \times r_2$ $[r_3 = \tau_3 + f_1 \times f_2]$
- P_2^{-1} $[r_1 = \tau_1, r_2 = \tau_2]$

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 call four subroutines: P1 and P2 forward and backward. But, this algorithm is extremely efficient with memory. Notice that P1 and P2 can each use all of our memory. There is absolutely no memory that needs to be set aside for the parent routine's exclusive use. In the talk title, I mentioned "borrowing" memory. This is what I mean by that.

Now let's talk about how to apply these techniques to solving the Tree Evaluation Problem.

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.

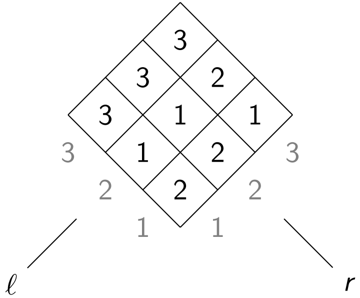
A formula for TEP

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. I'll introduce some notation: the *indicator* brackets x equals y is one if they are equal and otherwise zero.

A formula for TEP

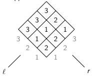
Let $R = \mathbb{Z}/2\mathbb{Z} = \{0, 1\}$. Define $[x = y] = 1$ if $x = y$, 0 otherwise.
Suppose node v has children ℓ and r:



$$[v = 1] =$$

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:

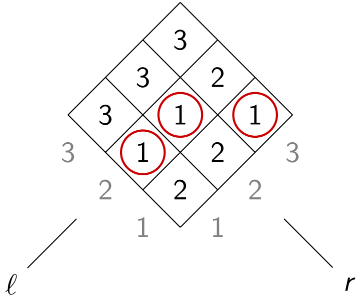


$[v = 1] =$

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

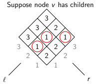
Let $R = \mathbb{Z}/2\mathbb{Z} = \{0, 1\}$. Define $[x = y] = 1$ if $x = y$, 0 otherwise.
Suppose node v has children ℓ and r:



$[v = 1] =$

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:

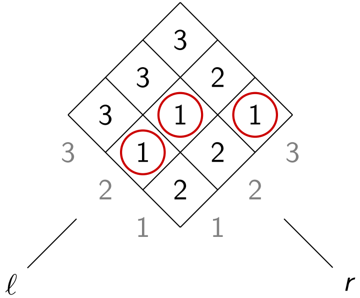


$[v = 1] =$

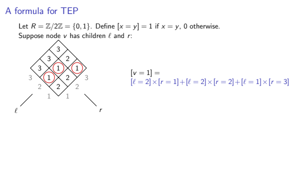
Well, there are three ways that node v can be equal to one, corresponding to the three one entries in the table. We can turn this into a formula with three terms, corresponding to these entries.

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:



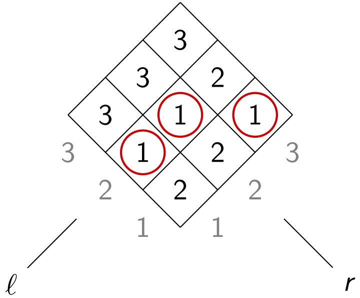
$[v = 1] =$
$[\ell = 2] \times [r = 1] + [\ell = 2] \times [r = 2] + [\ell = 1] \times [r = 3]$



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:



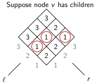
$$[v = 1] =$$
$$[\ell = 2] \times [r = 1] + [\ell = 2] \times [r = 2] + [\ell = 1] \times [r = 3]$$

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.

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 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 will be parameterized by a node v, a value x, and some target register i. The goal is to test whether v is equal to x. If they're equal, we flip the bit in register 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 lemma: 2 calls each to CheckNode(ℓ, y, j) and CheckNode(r, z, j'), where j and j' are two registers other than 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.
- 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 lemma: 2 calls each to CheckNode(ℓ, y, j) and CheckNode(r, z, j'), where j and j' are two registers other than i.

If v is an internal node, then we compute this formula by looping over all k squared possible values for y and z. In each case, the value we need to add depends on the product of the indicators ℓ equals y and r equals z. Using the technique I showed earlier for multiplication, this can be accomplished with four total 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 lemma: 2 calls each to CheckNode(ℓ, y, j) and CheckNode(r, z, j'), where j and j' are two registers other than i.

Needs three registers.

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 lemma: 2 calls each to CheckNode(ℓ, y, j) and CheckNode(r, z, j'), where j and j' are two registers other than i.

Needs three registers.

We set aside two other registers j and j' for the outputs of the recursive calls to CheckNode, because the multiplication algorithm needs that. That means we need a total of three registers: i, j and j'. Since we're using clean computations, the calls to the subroutine are free to use those same three registers.

[You might be concerned that our product actually has three terms. But the third term, f v of y z equals x, doesn't cost us anything, since it's directly available in the input.]

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 lemma: 2 calls each to CheckNode(ℓ, y, j) and CheckNode(r, z, j'), where j and j' are two registers other than i.

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

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 lemma: 2 calls each to CheckNode(ℓ, y, j) and CheckNode(r, z, j'), where j and j' are two registers other than i.

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

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 k squared to the power h minus one, since at every level, we make 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 lemma: 2 calls each to CheckNode(ℓ, y, j) and CheckNode(r, z, j'), where j and j' are two registers other than i.

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

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 lemma: 2 calls each to CheckNode(ℓ, y, j) and CheckNode(r, z, j'), where j and j' are two registers other than i.

Needs three registers. Gives branching program with width 8 and length $(k^2)^{h-1}$.
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 $(y, z) \in [k]^2$:
  - $r_i \leftarrow r_i + [f_v(y, z) = x] \times [\ell = y] \times [r = z]$

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

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

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

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.
Instead of three registers, we're going to end up needing 3 times k registers, since we're dealing with k possible values for the left child, k possible values for the right child, and, to complete the recursion, we'll need to provide k different output values as well.
We can think of the output of the subroutine as computing a k-bit string, where exactly one of the bits is one and the others are zero. We call this a *one-hot encoding*.

One-hot encoding

Given a value $x \in [k]$, define $\mathrm{OneHot}(x) = ([x = 1], [x = 2], \ldots, [x = k]) \in \{0, 1\}^k$.

E.g. for $k = 3$, $\mathrm{OneHot}(2) = (0, 1, 0)$.

One-hot encoding

Given a value $x \in [k]$, define $\text{OneHot}(x) = ([x = 1], [x = 2], \ldots, [x = k]) \in \{0, 1\}^k$.
E.g. for $k = 3$, $\text{OneHot}(2) = (0, 1, 0)$.

In other words, the one-hot encoding of x is a vector consisting of the indicators x equals 1 through x equals k. For example, when k is three, the one-hot encoding of 2 is zero one zero. So, let's see an algorithm to compute this.

Algorithm

ComputeOneHot(v, i) Uses *vector registers* $\vec{r_i} \in \{0, 1\}^k$.

Parameters: node v, target register i

Computes $\vec{r_i} \leftarrow \vec{r_i} + \text{OneHot}(v)$

Algorithm

ComputeOneHot(v, i) Uses *vector registers* $\vec{r_i} \in \{0,1\}^k$.
Parameters: node v, target register i
Computes $\vec{r_i} \leftarrow \vec{r_i} + \text{OneHot}(v)$

For this algorithm, the registers will store vectors of k bits each.
The algorithm is parameterized by a node v and a target register for the output. The goal is to flip exactly one bit of that register, where the index of the flipped bit is the value v.

Algorithm

ComputeOneHot(v, i) Uses *vector registers* $\vec{r_i} \in \{0,1\}^k$.

Parameters: node v, target register i

Computes $\vec{r_i} \leftarrow \vec{r_i} + \text{OneHot}(v)$

- If v is a leaf:
    - $\vec{r_i} \leftarrow \vec{r_i} + \text{OneHot}(v)$ is one instruction.

Algorithm

ComputeOneHot(v, i) Uses *vector registers* $\vec{r_i} \in \{0,1\}^k$.

Parameters: node v, target register i

Computes $\vec{r_i} \leftarrow \vec{r_i} + \text{OneHot}(v)$

- If v is a leaf:
  - $\vec{r_i} \leftarrow \vec{r_i} + \text{OneHot}(v)$ is one instruction.

If v is a leaf node, then we can still do this in one instruction, since the branching program has direct access to the leaf value.

Algorithm

ComputeOneHot(v, i) Uses *vector registers* $\vec{r_i} \in \{0,1\}^k$.

Parameters: node v, target register i

Computes $\vec{r_i} \leftarrow \vec{r_i} + \mathrm{OneHot}(v)$

- If v is a leaf:
  - $\vec{r_i} \leftarrow \vec{r_i} + \mathrm{OneHot}(v)$ is one instruction.
- else:
  - $\vec{r_i} \leftarrow \vec{r_i} + F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_j} \leftarrow \vec{r_j} + \mathrm{OneHot}(\ell)$
  - $\vec{r_i} \leftarrow \vec{r_i} - F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_{j'}} \leftarrow \vec{r_{j'}} + \mathrm{OneHot}(r)$
  - $\vec{r_i} \leftarrow \vec{r_i} + F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_j} \leftarrow \vec{r_j} - \mathrm{OneHot}(\ell)$
  - $\vec{r_i} \leftarrow \vec{r_i} - F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_{j'}} \leftarrow \vec{r_{j'}} - \mathrm{OneHot}(r)$

Algorithm

ComputeOneHot(v, i) Uses *vector registers* $\vec{r_i} \in \{0,1\}^k$.
Parameters: node v, target register i
Computes $\vec{r_i} \leftarrow \vec{r_i} + \text{OneHot}(v)$

- If v is a leaf:
  - $\vec{r_i} \leftarrow \vec{r_i} + \text{OneHot}(v)$ is one instruction.
- else:
  - $\vec{r_i} \leftarrow \vec{r_i} + F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_j} \leftarrow \vec{r_j} + \text{OneHot}(\ell)$
  - $\vec{r_i} \leftarrow \vec{r_i} - F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_{j'}} \leftarrow \vec{r_{j'}} + \text{OneHot}(r)$
  - $\vec{r_i} \leftarrow \vec{r_i} + F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_j} \leftarrow \vec{r_j} - \text{OneHot}(\ell)$
  - $\vec{r_i} \leftarrow \vec{r_i} - F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_{j'}} \leftarrow \vec{r_{j'}} - \text{OneHot}(r)$

When v is an internal node, we run our multiplication algorithm. As usual, this requires four recursive calls, except this time, each call is cleanly computing a k-bit vector of values. I've again marked the recursive calls in blue.

Just like before, the recursive calls can make full use of all of the registers, as long as they restore them when they're done. So, this whole algorithm only uses a total of three vector registers, for a total of three k bits of memory.

In between the recursive calls, we do some operations to mix together the results of the recursive calls. In the multiplication lemma, this would involve adding and subtracting the product r j times r j prime. Here, instead, we add and subtract some function capital F.

Algorithm

ComputeOneHot(v, i) Uses *vector registers* $\vec{r_i} \in \{0,1\}^k$.

Parameters: node v, target register i

Computes $\vec{r_i} \leftarrow \vec{r_i} + \text{OneHot}(v)$

- If v is a leaf:
  - $\vec{r_i} \leftarrow \vec{r_i} + \text{OneHot}(v)$ is one instruction.
- else:
  - $\vec{r_i} \leftarrow \vec{r_i} + F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_j} \leftarrow \vec{r_j} + \text{OneHot}(\ell)$
  - $\vec{r_i} \leftarrow \vec{r_i} - F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_{j'}} \leftarrow \vec{r_{j'}} + \text{OneHot}(r)$
  - $\vec{r_i} \leftarrow \vec{r_i} + F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_j} \leftarrow \vec{r_j} - \text{OneHot}(\ell)$
  - $\vec{r_i} \leftarrow \vec{r_i} - F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_{j'}} \leftarrow \vec{r_{j'}} - \text{OneHot}(r)$

$$F(\vec{r_j}, \vec{r_{j'}})_x = \sum_{(y,z)\in[k]^2} [f_v(y,z) = x] \times (\vec{r_j})_y \times (\vec{r_{j'}})_z$$

Note:

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

Algorithm

ComputeOneHot(v, i) Uses *vector registers* $\vec{r_i} \in \{0,1\}^k$.
Parameters: node v, target register i
Computes $\vec{r_i} \leftarrow \vec{r_i} + \text{OneHot}(v)$

- If v is a leaf:
  - $\vec{r_i} \leftarrow \vec{r_i} + \text{OneHot}(v)$ is one instruction.
- else:
  - $\vec{r_i} \leftarrow \vec{r_i} + F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_j} \leftarrow \vec{r_j} + \text{OneHot}(\ell)$
  - $\vec{r_i} \leftarrow \vec{r_i} - F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_{j'}} \leftarrow \vec{r_{j'}} + \text{OneHot}(r)$
  - $\vec{r_i} \leftarrow \vec{r_i} + F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_j} \leftarrow \vec{r_j} - \text{OneHot}(\ell)$
  - $\vec{r_i} \leftarrow \vec{r_i} - F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_{j'}} \leftarrow \vec{r_{j'}} - \text{OneHot}(r)$

$$F(\vec{r_j}, \vec{r_{j'}})_x = \sum_{(y,z)\in[k]^2} [f_v(y,z) = x] \times (\vec{r_j})_y \times (\vec{r_{j'}})_z$$

Note:

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

Our definition of capital F goes back to the original formula we computed for the indicator v equals x. To compute coordinate number x of that function, we take our original formula, and replace the indicator ℓ equals y with the y-th coordinate of register j, and the indicator r equals z with the z-th coordinate of register j prime. The net effect is that we are computing all of the products in this formula in parallel.

Now, if we turn this into a branching program, how many states does it use?

Algorithm

ComputeOneHot(v, i) Uses *vector registers* $\vec{r_i} \in \{0,1\}^k$.

Parameters: node v, target register i

Computes $\vec{r_i} \leftarrow \vec{r_i} + \text{OneHot}(v)$

- If v is a leaf:
  - $\vec{r_i} \leftarrow \vec{r_i} + \text{OneHot}(v)$ is one instruction.
- else:
  - $\vec{r_i} \leftarrow \vec{r_i} + F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_j} \leftarrow \vec{r_j} + \text{OneHot}(\ell)$
  - $\vec{r_i} \leftarrow \vec{r_i} - F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_{j'}} \leftarrow \vec{r_{j'}} + \text{OneHot}(r)$
  - $\vec{r_i} \leftarrow \vec{r_i} + F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_j} \leftarrow \vec{r_j} - \text{OneHot}(\ell)$
  - $\vec{r_i} \leftarrow \vec{r_i} - F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_{j'}} \leftarrow \vec{r_{j'}} - \text{OneHot}(r)$

$$F(\vec{r_j}, \vec{r_{j'}})_x = \sum_{(y,z)\in[k]^2} [f_v(y,z) = x] \times (\vec{r_j})_y \times (\vec{r_{j'}})_z$$

Note:

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

Gives branching program with width 2^{3k}, length $\Theta(k^2 4^h)$. Total $2^{\Theta(k+h)}$ states.

Algorithm

ComputeOneHot(v, i) Uses *vector registers* $\vec{r_i} \in \{0,1\}^k$.
Parameters: node v, target register i
Computes $\vec{r_i} \leftarrow \vec{r_i} + \text{OneHot}(v)$

- If v is a leaf:
  - $\vec{r_i} \leftarrow \vec{r_i} + \text{OneHot}(v)$ is one instruction.
- else:
  - $\vec{r_i} \leftarrow \vec{r_i} + F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_j} \leftarrow \vec{r_j} + \text{OneHot}(\ell)$
  - $\vec{r_i} \leftarrow \vec{r_i} - F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_{j'}} \leftarrow \vec{r_{j'}} + \text{OneHot}(r)$
  - $\vec{r_i} \leftarrow \vec{r_i} + F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_j} \leftarrow \vec{r_j} - \text{OneHot}(\ell)$
  - $\vec{r_i} \leftarrow \vec{r_i} - F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_{j'}} \leftarrow \vec{r_{j'}} - \text{OneHot}(r)$

$$F(\vec{r_j}, \vec{r_{j'}})_x = \sum_{(y,z)\in[k]^2} [f_v(y,z) = x] \times (\vec{r_j})_y \times (\vec{r_{j'}})_z$$

Note:

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

Gives branching program with width 2^{3k}, length $\Theta(k^2 4^h)$. Total $2^{\Theta(k+h)}$ states.

We have three k-bit registers, so our branching program will have width two to the three k. We make four recursive calls at each level, for a total of four to the h minus one. We need to do on the order of k squared work to compute the function captial F, so the total length is order k squared four to the h. Putting it all together, we get two to the big theta of k plus h states. Let's compare that to the original pebbling algorithm.

Pebbling algorithm: $\Theta((k+1)^h)$
ComputeOneHot: $2^{\Theta(k+h)}$ states.

Pebbling algorithm: $\Theta((k+1)^h)$
ComputeOneHot: $2^{\Theta(k+h)}$ states.

The Pebbling algorithm uses on the order of k plus one to the h states. This new algorithm uses two to the order k plus h states.

Pebbling algorithm: $\Theta((k+1)^h) = \Theta(2^{h \log_2(k+1)})$
ComputeOneHot: $2^{\Theta(k+h)}$ states. Better when $h \log(k+1) >> k+h$, i.e. when $h >> \dfrac{k}{\log k}$.

Pebbling algorithm: $\Theta((k+1)^h) = \Theta(2^{h \log_2 (k+1)})$
ComputeOneHot: $2^{\Theta(k+h)}$ states. Better when $h \log(k+1) >> k + h$, i.e. when $h >> \frac{k}{\log k}$.

We can write k plus one to the h as two to the h log k plus one. So, the one-hot algorithm is better when h log k plus one grows faster than k plus h. In other words, when the height h grows faster than k over log k.

Pebbling algorithm: $\Theta((k+1)^h) = \Theta(2^{h \log_2(k+1)})$
ComputeOneHot: $2^{\Theta(k+h)}$ states. Better when $h \log(k+1) >> k + h$, i.e. when $h >> \dfrac{k}{\log k}$.

Can we do better?

Pebbling algorithm: $\Theta((k+1)^h) = \Theta(2^{h \log_2(k+1)})$
ComputeOneHot: $2^{\Theta(k+h)}$ states. Better when $h \log(k+1) >> k+h$, i.e. when $h >> \frac{k}{\log k}$.

Can we do better?

Of course the next question is: can we improve on this?
The answer is yes, but it will take us a couple of steps to get there.
The problem with the one-hot algorithm is that its encoding is really inefficient. We're using k bits of memory to store something that could be stored in log k bits. So, let's see what happens if we switch to a binary encoding.

Binary encoding

Given a value $x \in [k]$, let $\text{Bin}(x) \in \{0,1\}^{\lceil \log k \rceil}$ be its binary encoding.

E.g. for $k = 3$, $\text{Bin}(1) = (0, 1)$.

Binary encoding

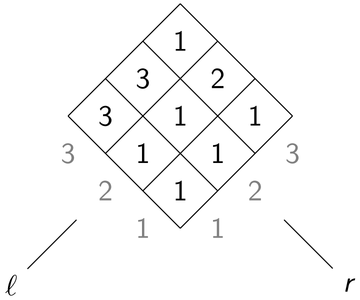
Given a value $x \in [k]$, let $\mathrm{Bin}(x) \in \{0,1\}^{\lceil \log k \rceil}$ be its binary encoding.
E.g. for $k = 3$, $\mathrm{Bin}(1) = (0,1)$.

The binary encoding produces a vector of length ceiling of log k. For example, when k is 3, the binary encoding of 1 is zero one.
The starting point for our one-hot algorithm was a formula for computing one bit of the one-hot encoding of the root node, based on the one-hot encodings of its children. Let's try to do the same thing here.

Binary encoding

Given a value $x \in [k]$, let $\mathrm{Bin}(x) \in \{0,1\}^{\lceil \log k \rceil}$ be its binary encoding.

E.g. for $k = 3$, $\mathrm{Bin}(1) = (0,1)$.

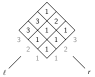


$$\mathrm{Bin}(v)_1 = [v = 2] + [v = 3] =$$

Binary encoding

Given a value $x \in [k]$, let $\mathrm{Bin}(x) \in \{0,1\}^{\lceil \log k \rceil}$ be its binary encoding.

E.g. for $k = 3$, $\mathrm{Bin}(1) = (0,1)$.



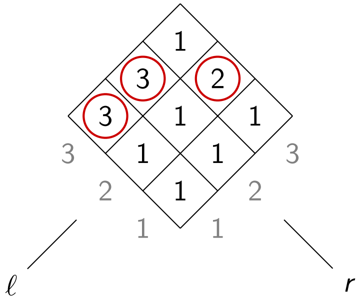
$\mathrm{Bin}(v)_1 = [v = 2] + [v = 3] =$

Let's say our root node has this table in it, and we're trying to compute the twos bit of its binary encoding, based on the values of its children ℓ and r. The twos bit is one when v is equal to two or three.

Binary encoding

Given a value $x \in [k]$, let $\text{Bin}(x) \in \{0,1\}^{\lceil \log k \rceil}$ be its binary encoding.

E.g. for $k = 3$, $\text{Bin}(1) = (0,1)$.

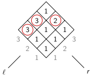


$$\text{Bin}(v)_1 = [v = 2] + [v = 3] =$$

Binary encoding

Given a value $x \in [k]$, let $\mathrm{Bin}(x) \in \{0,1\}^{\lceil \log k \rceil}$ be its binary encoding.

E.g. for $k = 3$, $\mathrm{Bin}(1) = (0,1)$.



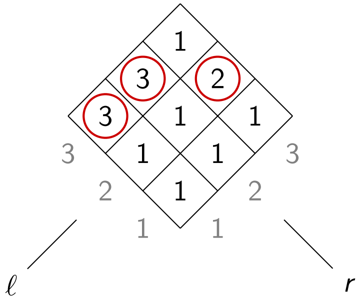
$\mathrm{Bin}(v)_1 = [v = 2] + [v = 3] =$

There are three entries in the table where 2 and 3 appear. We can turn this into a formula, just like before.

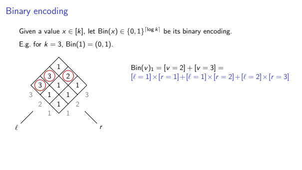
Binary encoding

Given a value $x \in [k]$, let $\text{Bin}(x) \in \{0,1\}^{\lceil \log k \rceil}$ be its binary encoding.

E.g. for $k = 3$, $\text{Bin}(1) = (0,1)$.



$\text{Bin}(v)_1 = [v = 2] + [v = 3] =$
$[\ell = 1] \times [r = 1] + [\ell = 1] \times [r = 2] + [\ell = 2] \times [r = 3]$



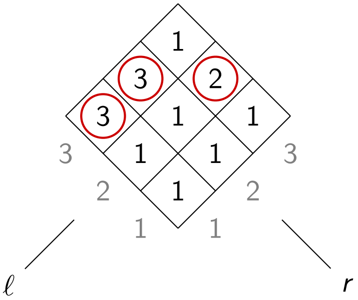
Our formula has three terms, corresponding to these three locations: either ℓ and r equal 1, or ℓ equals 1 and r equals 2, or ℓ equals 2 and r equals 3.

This formula is written in terms of indicators. If we're going to apply this recursively, we need to rewrite the formula in terms of the binary encoding.

Binary encoding

Given a value $x \in [k]$, let $\mathrm{Bin}(x) \in \{0,1\}^{\lceil \log k \rceil}$ be its binary encoding.

E.g. for $k = 3$, $\mathrm{Bin}(1) = (0, 1)$.



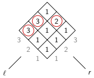
$\mathrm{Bin}(v)_1 = [v = 2] + [v = 3] =$
$[\ell = 1] \times [r = 1] + [\ell = 1] \times [r = 2] + [\ell = 2] \times [r = 3]$

$[\ell = 1] = (1 + \mathrm{Bin}(\ell)_1) \times \mathrm{Bin}(\ell)_2$

Binary encoding

Given a value $x \in [k]$, let $\mathrm{Bin}(x) \in \{0,1\}^{\lceil \log k \rceil}$ be its binary encoding.

E.g. for $k = 3$, $\mathrm{Bin}(1) = (0,1)$.



$\mathrm{Bin}(v)_1 = [v = 2] + [v = 3] =$
$[\ell = 1] \times [r = 1] + [\ell = 1] \times [r = 2] + [\ell = 2] \times [r = 3]$

$[\ell = 1] = (1 + \mathrm{Bin}(\ell)_1) \times \mathrm{Bin}(\ell)_2$

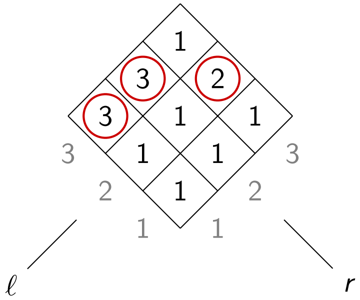
ℓ is equal to one when the first bit of its binary encoding is zero and the second bit is one. We can write this as: one plus the first bit times the second bit. Remember that we're working mod two, so one plus flips the bit.

Now, let's go back and expand the whole original formula this way.

Binary encoding

Given a value $x \in [k]$, let $\text{Bin}(x) \in \{0,1\}^{\lceil \log k \rceil}$ be its binary encoding.

E.g. for $k = 3$, $\text{Bin}(1) = (0,1)$.



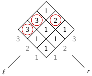
$\text{Bin}(v)_1 = [v=2] + [v=3] =$
$[\ell=1]\times[r=1] + [\ell=1]\times[r=2] + [\ell=2]\times[r=3]$
$= (1+\text{Bin}(\ell)_1) \times \text{Bin}(\ell)_2 \times (1+\text{Bin}(r)_1) \times \text{Bin}(r)_2$
$+ (1+\text{Bin}(\ell)_1) \times \text{Bin}(\ell)_2 \times \text{Bin}(r)_1 \times (1+\text{Bin}(r)_2)$
$+ \text{Bin}(\ell)_1 \times (1+\text{Bin}(\ell)_2) \times \text{Bin}(r)_1 \times \text{Bin}(r)_2$

$[\ell = 1] = (1+\text{Bin}(\ell)_1) \times \text{Bin}(\ell)_2$

Binary encoding

Given a value $x \in [k]$, let $\text{Bin}(x) \in \{0,1\}^{\lceil \log k \rceil}$ be its binary encoding.

E.g. for $k = 3$, $\text{Bin}(1) = (0,1)$.



$\text{Bin}(v)_1 = [v = 2] + [v = 3] =$
$[\ell = 1] \times [r = 1] + [\ell = 1] \times [r = 2] + [\ell = 2] \times [r = 3]$
$= (1 + \text{Bin}(\ell)_1) \times \text{Bin}(\ell)_2 \times (1 + \text{Bin}(r)_1) \times \text{Bin}(r)_2$
$+ (1 + \text{Bin}(\ell)_1) \times \text{Bin}(\ell)_2 \times \text{Bin}(r)_1 \times (1 + \text{Bin}(r)_2)$
$+ \text{Bin}(\ell)_1 \times (1 + \text{Bin}(\ell)_2) \times \text{Bin}(r)_1 \times \text{Bin}(r)_2$

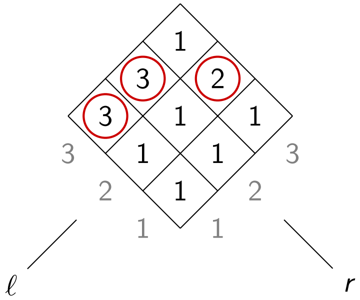
$[\ell = 1] = (1 + \text{Bin}(\ell)_1) \times \text{Bin}(\ell)_2$

We get a sum of three terms, and each term is a product of four different bits. Some of the bits are negated by taking one minus the bit.

Binary encoding

Given a value $x \in [k]$, let $\mathrm{Bin}(x) \in \{0,1\}^{\lceil \log k \rceil}$ be its binary encoding.

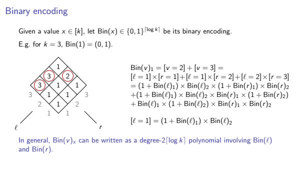
E.g. for $k = 3$, $\mathrm{Bin}(1) = (0,1)$.



$$\mathrm{Bin}(v)_1 = [v=2] + [v=3] =$$
$$[\ell=1]\times[r=1] + [\ell=1]\times[r=2] + [\ell=2]\times[r=3]$$
$$= (1+\mathrm{Bin}(\ell)_1)\times \mathrm{Bin}(\ell)_2 \times (1+\mathrm{Bin}(r)_1)\times \mathrm{Bin}(r)_2$$
$$+(1+\mathrm{Bin}(\ell)_1)\times \mathrm{Bin}(\ell)_2 \times \mathrm{Bin}(r)_1 \times (1+\mathrm{Bin}(r)_2)$$
$$+\mathrm{Bin}(\ell)_1 \times (1+\mathrm{Bin}(\ell)_2)\times \mathrm{Bin}(r)_1 \times \mathrm{Bin}(r)_2$$

$$[\ell=1] = (1+\mathrm{Bin}(\ell)_1)\times \mathrm{Bin}(\ell)_2$$

In general, $\mathrm{Bin}(v)_x$ can be written as a degree-$2\lceil \log k \rceil$ polynomial involving $\mathrm{Bin}(\ell)$ and $\mathrm{Bin}(r)$.



In general, any bit of the binary encoding can be computed using a polynomial with degree two log k involving the bits of ℓ and r.

Our multiplication lemma from earlier can only handle two inputs at a time. Since the polynomial has degree more than two, we're going to need to upgrade our lemma.

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}
- $r_3 \leftarrow r_3 + r_1 \times r_2$

Here's our original lemma. Remember, we assume we have subroutines P1 and P2 that cleanly compute functions f1 and f2.

The way the program worked was by alternating between P1 and P2, and doing some fiddling in between each step.

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:

- $P_1 \qquad [r_1 = \tau_1 + f_1, r_2 = \tau_2]$
- $r_3 \leftarrow r_3 - r_1 \times r_2$
- $P_2 \qquad [r_1 = \tau_1 + f_1, r_2 = \tau_2 + f_2]$
- $r_3 \leftarrow r_3 + r_1 \times r_2$
- $P_1^{-1} \qquad [r_1 = \tau_1, r_2 = \tau_2 + f_2]$
- $r_3 \leftarrow r_3 - r_1 \times r_2$
- $P_2^{-1} \qquad [r_1 = \tau_1, r_2 = \tau_2]$
- $r_3 \leftarrow r_3 + r_1 \times r_2$

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 $[r_1 = \tau_1 + f_1, r_2 = \tau_2]$
- $r_3 \leftarrow r_3 - r_1 \times r_2$
- P_2 $[r_1 = \tau_1 + f_1, r_2 = \tau_2 + f_2]$
- $r_3 \leftarrow r_3 + r_1 \times r_2$
- P_1^{-1} $[r_1 = \tau_1, r_2 = \tau_2 + f_2]$
- $r_3 \leftarrow r_3 - r_1 \times r_2$
- P_2^{-1} $[r_1 = \tau_1, r_2 = \tau_2]$
- $r_3 \leftarrow r_3 + r_1 \times r_2$

The important thing is that after every subroutine call, we have a different combination loaded into registers r1 and r2. r1 can either be tau1 or tau1 plus f1, and r2 can either be tau2 or tau2 plus r2, and all four combinations appear.
Generalizing this is going to involve a loop over all subsets.

Lemma: d-ary multiplication

Suppose we have d values $f_1, \ldots, f_d$, and a general subroutine P. For any $S \subseteq [d]$, $P(S)$ cleanly computes $r_i \leftarrow r_i + f_i$ for every $i \in S$, and leaves r_j alone for $j \notin S$. Then we can cleanly compute $f_1 \times \cdots \times f_d$ into r_{d+1} as follows:

Lemma: *d*-ary multiplication

Suppose we have d values $f_1, \dots, f_d$, and a general subroutine P. For any $S \subseteq [d]$, $P(S)$ cleanly computes $r_i \leftarrow r_i + f_i$ for every $i \in S$, and leaves r_j alone for $j \notin S$. Then we can cleanly compute $f_1 \times \cdots \times f_d$ into r_{d+1} as follows:

So, here's the setup. We have d values f1 through fd. We have a subroutine P that can compute any combination of them. For any set S, P of S cleanly computes fi into register i for all i in S, and doesn't touch the remaining registers.
Given that, we can cleanly compute the product of all the fs into register d plus one like this.

Lemma: d-ary multiplication

Suppose we have d values $f_1, \ldots, f_d$, and a general subroutine P. For any $S \subseteq [d]$, $P(S)$ cleanly computes $r_i \leftarrow r_i + f_i$ for every $i \in S$, and leaves r_j alone for $j \notin S$. Then we can cleanly compute $f_1 \times \cdots \times f_d$ into r_{d+1} as follows:

- For every subset $S \subseteq [d]$:
  - Call $P(S')$, choosing S' so that $r_i = \tau_i$ for $i \notin S$, and $r_i = \tau_i + f_i$ for $i \in S$.
  - $r_{d+1} \leftarrow r_{d+1} + c_S \times \prod_{i=1}^{d} r_i$
- Call P once more to ensure $r_i = \tau_i$ for $i = 1, \ldots, d$.

Lemma: *d*-ary multiplication

Suppose we have d values $f_1, \ldots, f_d$, and a general subroutine P. For any $S \subseteq [d]$, $P(S)$ cleanly computes $r_i \leftarrow r_i + f_i$ for every $i \in S$, and leaves r_j alone for $j \notin S$. Then we can cleanly compute $f_1 \times \cdots \times f_d$ into r_{d+1} as follows:

- For every subset $S \subseteq [d]$:
  - Call $P(S')$, choosing S' so that $r_i = \tau_i$ for $i \notin S$, and $r_i = \tau_i + f_i$ for $i \in S$.
  - $r_{d+1} \leftarrow r_{d+1} + c_S \times \prod_{i=1}^{d} r_i$
- Call P once more to ensure $r_i = \tau_i$ for $i = 1, \ldots, d$.

We loop through all possible sets of values. For each set S, we set up the registers r1 through rd so that register i either has its original value or its original value plus f i, depending on the set S. We can do this by crafting a set S' which lists all of the registers we want to toggle between holding their original value and original value plus f i.
Then we add the product of registers 1 through d, times a carefully crafted constant c sub S.
After we've done this for every possible subset, we call P one last time to restore the registers r1 through r d to their original values, in order to make this a clean computation.

Lemma: d-ary multiplication

Suppose we have d values $f_1, \dots, f_d$, and a general subroutine P. For any $S \subseteq [d]$, $P(S)$ cleanly computes $r_i \leftarrow r_i + f_i$ for every $i \in S$, and leaves r_j alone for $j \notin S$. Then we can cleanly compute $f_1 \times \cdots \times f_d$ into r_{d+1} as follows:

- For every subset $S \subseteq [d]$:
  - Call $P(S')$, choosing S' so that $r_i = \tau_i$ for $i \notin S$, and $r_i = \tau_i + f_i$ for $i \in S$.
  - $r_{d+1} \leftarrow r_{d+1} + c_S \times \prod_{i=1}^{d} r_i$
- Call P once more to ensure $r_i = \tau_i$ for $i = 1, \dots, d$.

Uses $d + 1$ registers and 2^d recursive calls.

Lemma: d-ary multiplication

Suppose we have d values $f_1, \dots, f_d$, and a general subroutine P. For any $S \subseteq [d]$, $P(S)$ cleanly computes $r_i \leftarrow r_i + f_i$ for every $i \in S$, and leaves r_j alone for $j \notin S$. Then we can cleanly compute $f_1 \times \cdots \times f_d$ into r_{d+1} as follows:

- For every subset $S \subseteq [d]$:
  - Call $P(S')$, choosing S' so that $r_i = \tau_i$ for $i \notin S$, and $r_i = \tau_i + f_i$ for $i \in S$.
  - $r_{d+1} \leftarrow r_{d+1} + c_S \times \prod_{i=1}^{d} r_i$
- Call P once more to ensure $r_i = \tau_i$ for $i = 1, \dots, d$.

Uses $d + 1$ registers and 2^d recursive calls.

This algorithm uses d plus 1 registers and two to the d recursive calls.

Let's try using it in an algorithm. Notice that the subroutine is expected to be able to compute any subset S of the bits 1 through k. So, S should be a parameter for our algorithm.

Algorithm

ComputeBin(v, S, i)   Uses *vector registers* $\vec{r_i} \in \{0,1\}^{\lceil \log k \rceil}$.

Parameters: node v, set $S \subseteq [\log k]$, target register i

Computes: $\vec{r_i}_b \leftarrow \vec{r_i}_b + \text{Bin}(v)_b$ for all $b \in S$

Algorithm

ComputeBin(v, S, i) Uses *vector registers* $\vec{r_i} \in \{0,1\}^{\lceil \log k \rceil}$.
Parameters: node v, set $S \subseteq [\log k]$, target register i
Computes: $\vec{r_i}_b \leftarrow \vec{r_i}_b + \text{Bin}(v)_b$ for all $b \in S$

The binary algorithm uses registers that each store log k bits.
As before, it's parameterized by a node v and a target register for the output, but now we've added the set S, telling us which bits should be computed.
The goal is to flip all of the bits that are in S and correspond to ones in the binary encoding of v.
For example, if v is zero, or if the set S is empty, then this subroutine should make no changes at all.

Algorithm

ComputeBin(v, S, i) Uses *vector registers* $\vec{r_i} \in \{0,1\}^{\lceil \log k \rceil}$.

Parameters: node v, set $S \subseteq [\log k]$, target register i

Computes: $\vec{r_i}_b \leftarrow \vec{r_i}_b + \text{Bin}(v)_b$ for all $b \in S$

- If v is a leaf:
  - $\vec{r_i} \leftarrow \vec{r_i} + \text{ComputeBin}(v)$ is one instruction.

Algorithm

ComputeBin(v, S, i) Uses *vector registers* $\vec{r_i} \in \{0,1\}^{\lceil \log k \rceil}$.
Parameters: node v, set $S \subseteq [\log k]$, target register i
Computes: $\vec{r_i}_b \leftarrow \vec{r_i}_b + \text{Bin}(v)_b$ for all $b \in S$

- If v is a leaf:
  - $\vec{r_i} \leftarrow \vec{r_i} + \text{ComputeBin}(v)$ is one instruction.

As before, if v is a leaf node, one instruction is enough.

Algorithm

ComputeBin(v, S, i) Uses *vector registers* $\vec{r_i} \in \{0,1\}^{\lceil \log k \rceil}$.

Parameters: node v, set $S \subseteq [\log k]$, target register i

Computes: $\vec{r_i}_b \leftarrow \vec{r_i}_b + \text{Bin}(v)_b$ for all $b \in S$

- If v is a leaf:
  - $\vec{r_i} \leftarrow \vec{r_i} + \text{ComputeBin}(v)$ is one instruction.
- else:
  - for all subsets $T_1, T_2 \subseteq [\log k]$:
    - Call ComputeBin(ℓ, T_1', j) and ComputeBin(r, T_2', j').
    - for all $b \in S$, $(\vec{r_i})_b \leftarrow (\vec{r_i})_b + F(\vec{r_j}, \vec{r_{j'}})$

Algorithm

ComputeBin(v, S, i) Uses *vector registers* $\vec{r_i} \in \{0,1\}^{\lceil \log k \rceil}$.
Parameters: node v, set $S \subseteq [\log k]$, target register i
Computes: $\vec{r_i}_b \leftarrow \vec{r_i}_b + \text{Bin}(v)_b$ for all $b \in S$

- If v is a leaf:
  - $\vec{r_i} \leftarrow \vec{r_i} + \text{ComputeBin}(v)$ is one instruction.
- else:
  - for all subsets $T_1, T_2 \subseteq [\log k]$:
    - Call ComputeBin(ℓ, T_1', j) and ComputeBin(r, T_2', j').
    - for all $b \in S$, $(\vec{r_i})_b \leftarrow (\vec{r_i})_b + F(\vec{r_j}, \vec{r_{j'}})$

When v is an internal node, we run our d-ary multiplication algorithm in parallel for all of the bits b in S.

That means we loop through all possible subsets T_1 and T_2 of the log k bits.

After each call, we update our output registers according to some function F. I don't have F on the slide, but it's a big polynomial that relates the binary encoding of v to the binary encodings of its children.

Now, if we turn this into a branching program, let's figure out how many states it uses.

Algorithm

ComputeBin(v, S, i) Uses *vector registers* $\vec{r_i} \in \{0,1\}^{\lceil \log k \rceil}$.

Parameters: node v, set $S \subseteq [\log k]$, target register i

Computes: $\vec{r_i}_b \leftarrow \vec{r_i}_b + \mathrm{Bin}(v)_b$ for all $b \in S$

- If v is a leaf:
  - $\vec{r_i} \leftarrow \vec{r_i} + \mathrm{ComputeBin}(v)$ is one instruction.
- else:
  - for all subsets $T_1, T_2 \subseteq [\log k]$:
    - Call ComputeBin(ℓ, T_1', j) and ComputeBin(r, T_2', j').
    - for all $b \in S$, $(\vec{r_i})_b \leftarrow (\vec{r_i})_b + F(\vec{r_j}, \vec{r_{j'}})$

ComputeBin uses $3 \log k$ bits of memory and makes $2 \times 2^{2 \log k} = 2k^2$ recursive calls.

Algorithm

ComputeBin(v, S, i)   Uses *vector registers* $\vec{r_i} \in \{0,1\}^{\lceil \log k \rceil}$.

Parameters: node v, set $S \subseteq [\log k]$, target register i

Computes: $\vec{r_i}_b \leftarrow \vec{r_i}_b + \text{Bin}(v)_b$ for all $b \in S$

- If v is a leaf:
  - $\vec{r_i} \leftarrow \vec{r_i} + \text{ComputeBin}(v)$ is one instruction.
- else:
  - for all subsets $T_1, T_2 \subseteq [\log k]$:
    - Call ComputeBin(ℓ, T_1', j) and ComputeBin(r, T_2', j').
    - for all $b \in S$, $(\vec{r_i})_b \leftarrow (\vec{r_i})_b + F(\vec{r_j}, \vec{r_{j'}})$

ComputeBin uses $3 \log k$ bits of memory and makes $2 \times 2^{2 \log k} = 2k^2$ recursive calls.

Our algorithm uses three log k bits of memory. It loops over two to the power two log k different pairs of subsets, and makes two recursive calls for each one, which works out to 2 k squared recursive calls.

Algorithm

ComputeBin(v, S, i) Uses *vector registers* $\vec{r_i} \in \{0,1\}^{\lceil \log k \rceil}$.
Parameters: node v, set $S \subseteq [\log k]$, target register i
Computes: $\vec{r_i}_b \leftarrow \vec{r_i}_b + \text{Bin}(v)_b$ for all $b \in S$

- If v is a leaf:
  - $\vec{r_i} \leftarrow \vec{r_i} + \text{ComputeBin}(v)$ is one instruction.
- else:
  - for all subsets $T_1, T_2 \subseteq [\log k]$:
    - Call ComputeBin(ℓ, T_1', j) and ComputeBin(r, T_2', j').
    - for all $b \in S$, $(\vec{r_i})_b \leftarrow (\vec{r_i})_b + F(\vec{r_j}, \vec{r_{j'}})$

ComputeBin uses $3 \log k$ bits of memory and makes $2 \times 2^{2 \log k} = 2k^2$ recursive calls.
It gives branching program with width $2^{3 \log k} = k^3$ and length $\Theta((2k)^{2h} k^{O(1)})$. Total $\Theta(k^{2h+O(1)})$ states.

Algorithm

ComputeBin(v, S, i) Uses *vector registers* $\vec{r_i} \in \{0,1\}^{\lceil \log k \rceil}$.
Parameters: node v, set $S \subseteq [\log k]$, target register i
Computes: $\vec{r_i}_b \leftarrow \vec{r_i}_b + \mathrm{Bin}(v)_b$ for all $b \in S$

- If v is a leaf:
  - $\vec{r_i} \leftarrow \vec{r_i} + \mathrm{ComputeBin}(v)$ is one instruction.
- else:
  - for all subsets $T_1, T_2 \subseteq [\log k]$:
    - Call ComputeBin(ℓ, T_1', j) and ComputeBin(r, T_2', j').
    - for all $b \in S$, $(\vec{r_i})_b \leftarrow (\vec{r_i})_b + F(\vec{r_j}, \vec{r_{j'}})$

ComputeBin uses $3 \log k$ bits of memory and makes $2 \times 2^{2 \log k} = 2k^2$ recursive calls.
It gives branching program with width $2^{3 \log k} = k^3$ and length $\Theta((2k)^{2h} k^{O(1)})$. Total $\Theta(k^{2h+O(1)})$ states.

That works out to a width of k cubed states in each layer, and a length of 2k to the two h for h levels of recursive calls, times k to some constant power representing the work done to compute the function capital F.
Overall, that works out to on the order of k to the two h plus a constant states.

Algorithm

ComputeBin(v, S, i) Uses *vector registers* $\vec{r_i} \in \{0,1\}^{\lceil \log k \rceil}$.
Parameters: node v, set $S \subseteq [\log k]$, target register i
Computes: $\vec{r_i}_b \leftarrow \vec{r_i}_b + \mathrm{Bin}(v)_b$ for all $b \in S$

- If v is a leaf:
  - $\vec{r_i} \leftarrow \vec{r_i} + \mathrm{ComputeBin}(v)$ is one instruction.
- else:
  - for all subsets $T_1, T_2 \subseteq [\log k]$:
    - Call ComputeBin(ℓ, T_1', j) and ComputeBin(r, T_2', j').
    - for all $b \in S$, $(\vec{r_i})_b \leftarrow (\vec{r_i})_b + F(\vec{r_j}, \vec{r_{j'}})$

ComputeBin uses $3 \log k$ bits of memory and makes $2 \times 2^{2 \log k} = 2k^2$ recursive calls. It gives branching program with width $2^{3 \log k} = k^3$ and length $\Theta((2k)^{2h} k^{O(1)})$. Total $\Theta(k^{2h+O(1)})$ states.
Worse than pebbling, which uses $\Theta((k+1)^h)$ states.

Algorithm

ComputeBin(v, S, i) Uses *vector registers* $\vec{r_i} \in \{0,1\}^{\lceil \log k \rceil}$.
Parameters: node v, set $S \subseteq [\log k]$, target register i
Computes: $\vec{r_i}_b \leftarrow \vec{r_i}_b + \text{Bin}(v)_b$ for all $b \in S$

- If v is a leaf:
  - $\vec{r_i} \leftarrow \vec{r_i} + \text{ComputeBin}(v)$ is one instruction.
- else:
  - for all subsets $T_1, T_2 \subseteq [\log k]$:
    - Call ComputeBin(ℓ, T_1', j) and ComputeBin(r, T_2', j').
    - for all $b \in S$, $(\vec{r_i})_b \leftarrow (\vec{r_i})_b + F(\vec{r_j}, \vec{r_{j'}})$

ComputeBin uses $3 \log k$ bits of memory and makes $2 \times 2^{2 \log k} = 2k^2$ recursive calls.
It gives branching program with width $2^{3 \log k} = k^3$ and length $\Theta((2k)^{2h} k^{O(1)})$. Total $\Theta(k^{2h+O(1)})$ states.
Worse than pebbling, which uses $\Theta((k+1)^h)$ states.

Well, that means we're back to having an algorithm that's strictly worse than pebbling, which only uses k to the h states.
But this algorithm wasn't a waste of time. It turns out we can combine it with the one-hot algorithm to make a sort of hybrid that interpolates between both of them.
Let's take a step back and look at the algorithms we have so far.

algorithm	width	length	total states
One-hot	$2^{\Theta(k)}$	$\Theta(k^2 4^h)$	$2^{\Theta(k+h)}$
Binary	k^3	$k^{\Theta(h)}$	$k^{\Theta(h)}$

algorithm	width	length	total states
One-hot	$2^{\Theta(k)}$	$\Theta(k^2 4^h)$	$2^{\Theta(k+h)}$
Binary	k^3	$k^{\Theta(h)}$	$k^{\Theta(h)}$

Here are the two algorithms I've presented. In this table, width means the number of states in each layer of a branching program, and length is the number of layers.

The one-hot algorithm is very wide, but not too long. The one-hot encodings take up a lot of space, but we only need four recursive calls per layer.

The binary algorithm is the opposite. Its width is quite small, only k cubed. But it loses out on length, since it makes 2 k squared recursive calls at each layer.

The next algorithm I'll show you will combine these two.

algorithm	width	length	total states
One-hot	$2^{\Theta(k)}$	$\Theta(k^2 4^h)$	$2^{\Theta(k+h)}$
Binary	k^3	$k^{\Theta(h)}$	$k^{\Theta(h)}$
Hybrid	$2^{\Theta(\frac{a}{2^a-1}k)}$	$2^{\Theta(ah)}k^{\Theta(1)}$	$2^{\Theta(ah+\frac{a}{2^a-1}k))}$

algorithm	width	length	total states
One-hot	$2^{\Theta(k)}$	$\Theta(k^2 4^h)$	$2^{\Theta(k+h)}$
Binary	k^3	$k^{\Theta(h)}$	$k^{\Theta(h)}$
Hybrid	$2^{\Theta(\frac{a}{2^a-1}k)}$	$2^{\Theta(ah)}k^{\Theta(1)}$	$2^{\Theta(ah+\frac{a}{2^a-1}k))}$

We get to choose a parameter a. Let's see what happens if we choose large or small values for a.

algorithm	width	length	total states
One-hot	$2^{\Theta(k)}$	$\Theta(k^2 4^h)$	$2^{\Theta(k+h)}$
Binary	k^3	$k^{\Theta(h)}$	$k^{\Theta(h)}$
Hybrid	$2^{\Theta(\frac{a}{2^a-1}k)}$	$2^{\Theta(ah)} k^{\Theta(1)}$	$2^{\Theta(ah+\frac{a}{2^a-1}k))}$
Hybrid, $a = 1$	$2^{\Theta(k)}$	$2^{\Theta(h)} k^{\Theta(1)}$	$2^{\Theta(k+h)}$

algorithm	width	length	total states
One-hot	$2^{\Theta(k)}$	$\Theta(k^2 4^h)$	$2^{\Theta(k+h)}$
Binary	k^3	$k^{\Theta(h)}$	$k^{\Theta(h)}$
Hybrid	$2^{\Theta(\frac{a}{2^a-1}k)}$	$2^{\Theta(ah)}k^{\Theta(1)}$	$2^{\Theta(ah+\frac{a}{2^a-1}k))}$
Hybrid, $a=1$	$2^{\Theta(k)}$	$2^{\Theta(h)}k^{\Theta(1)}$	$2^{\Theta(k+h)}$

If we set a to be equal to one, then the width just becomes two to the theta k, and the length becomes two to the theta h times k to a constant. So, it's like the one-hot algorithm.

algorithm	width	length	total states
One-hot	$2^{\Theta(k)}$	$\Theta(k^2 4^h)$	$2^{\Theta(k+h)}$
Binary	k^3	$k^{\Theta(h)}$	$k^{\Theta(h)}$
Hybrid	$2^{\Theta(\frac{a}{2^a-1}k)}$	$2^{\Theta(ah)}k^{\Theta(1)}$	$2^{\Theta(ah+\frac{a}{2^a-1}k))}$
Hybrid, $a = 1$	$2^{\Theta(k)}$	$2^{\Theta(h)}k^{\Theta(1)}$	$2^{\Theta(k+h)}$
Hybrid, $a = \log(k+1)$	$k^{\Theta(1)}$	$k^{\Theta(h)}$	$k^{\Theta(h)}$

algorithm	width	length	total states
One-hot	$2^{\Theta(k)}$	$\Theta(k^2 4^h)$	$2^{\Theta(k+h)}$
Binary	k^3	$k^{\Theta(h)}$	$k^{\Theta(h)}$
Hybrid	$2^{\Theta(\frac{a}{2^a-1}k)}$	$2^{\Theta(ah)}k^{\Theta(1)}$	$2^{\Theta(ah+\frac{a}{2^a-1}k))}$
Hybrid, $a = 1$	$2^{\Theta(k)}$	$2^{\Theta(h)}k^{\Theta(1)}$	$2^{\Theta(k+h)}$
Hybrid, $a = \log(k+1)$	$k^{\Theta(1)}$	$k^{\Theta(h)}$	$k^{\Theta(h)}$

If we set a to the largest value, log k plus one, then the width shrinks down to k to a constant, and the length increases to two to the theta of h log k, which is k to the theta h. So, it becomes just like the binary algorithm.

algorithm	width	length	total states
One-hot	$2^{\Theta(k)}$	$\Theta(k^2 4^h)$	$2^{\Theta(k+h)}$
Binary	k^3	$k^{\Theta(h)}$	$k^{\Theta(h)}$
Hybrid	$2^{\Theta(\frac{a}{2^a-1}k)}$	$2^{\Theta(ah)}k^{\Theta(1)}$	$2^{\Theta(ah+\frac{a}{2^a-1}k))}$
Hybrid, $a = 1$	$2^{\Theta(k)}$	$2^{\Theta(h)}k^{\Theta(1)}$	$2^{\Theta(k+h)}$
Hybrid, $a = \log(k+1)$	$k^{\Theta(1)}$	$k^{\Theta(h)}$	$k^{\Theta(h)}$
Hybrid, $a = \log(\frac{k}{h}+1)$	$\Theta((\frac{k}{h}+1)^{\Theta(h)}$	$\Theta((\frac{k}{h}+1)^{\Theta(h)})$	$(\frac{k}{h}+1)^{5h}k^{\Theta(1)}$

algorithm	width	length	total states
One-hot	$2^{\Theta(k)}$	$\Theta(k^2 4^h)$	$2^{\Theta(k+h)}$
Binary	k^3	$k^{\Theta(h)}$	$k^{\Theta(h)}$
Hybrid	$2^{\Theta(\frac{a}{2^a-1}k)}$	$2^{\Theta(ah)}k^{\Theta(1)}$	$2^{\Theta(ah+\frac{a}{2^a-1}k))}$
Hybrid, $a=1$	$2^{\Theta(k)}$	$2^{\Theta(h)}k^{\Theta(1)}$	$2^{\Theta(k+h)}$
Hybrid, $a=\log(k+1)$	$k^{\Theta(1)}$	$k^{\Theta(h)}$	$k^{\Theta(h)}$
Hybrid, $a=\log(\frac{k}{h}+1)$	$\Theta((\frac{k}{h}+1)^{\Theta(h)}$	$\Theta((\frac{k}{h}+1)^{\Theta(h)})$	$(\frac{k}{h}+1)^{5h}k^{\Theta(1)}$

If we choose the parameter a carefully, we can get a good middle ground. If you look at the right-most column, it works out to k over h plus 1 to the power five h times k to a constant states.

algorithm	width	length	total states
One-hot	$2^{\Theta(k)}$	$\Theta(k^2 4^h)$	$2^{\Theta(k+h)}$
Binary	k^3	$k^{\Theta(h)}$	$k^{\Theta(h)}$
Hybrid	$2^{\Theta(\frac{a}{2^a-1}k)}$	$2^{\Theta(ah)}k^{\Theta(1)}$	$2^{\Theta(ah+\frac{a}{2^a-1}k))}$
Hybrid, $a = 1$	$2^{\Theta(k)}$	$2^{\Theta(h)}k^{\Theta(1)}$	$2^{\Theta(k+h)}$
Hybrid, $a = \log(k+1)$	$k^{\Theta(1)}$	$k^{\Theta(h)}$	$k^{\Theta(h)}$
Hybrid, $a = \log(\frac{k}{h}+1)$	$\Theta((\frac{k}{h}+1)^{\Theta(h)}$	$\Theta((\frac{k}{h}+1)^{\Theta(h)})$	$(\frac{k}{h}+1)^{5h}k^{\Theta(1)}$

Pebbling uses $\Theta((k+1)^h)$ states. Hybrid is better when $h = \omega(k^{4/5})$.

algorithm	width	length	total states
One-hot	$2^{\Theta(k)}$	$\Theta(k^2 4^h)$	$2^{\Theta(k+h)}$
Binary	k^3	$k^{\Theta(h)}$	$k^{\Theta(h)}$
Hybrid	$2^{\Theta(\frac{a}{2^a-1}k)}$	$2^{\Theta(ah)}k^{\Theta(1)}$	$2^{\Theta(ah+\frac{a}{2^a-1}k))}$
Hybrid, $a=1$	$2^{\Theta(k)}$	$2^{\Theta(h)}k^{\Theta(1)}$	$2^{\Theta(k+h)}$
Hybrid, $a=\log(k+1)$	$k^{\Theta(1)}$	$k^{\Theta(h)}$	$k^{\Theta(h)}$
Hybrid, $a=\log(\frac{k}{h}+1)$	$\Theta((\frac{k}{h}+1)^{\Theta(h)}$	$\Theta((\frac{k}{h}+1)^{\Theta(h)})$	$(\frac{k}{h}+1)^{5h}k^{\Theta(1)}$

Pebbling uses $\Theta((k+1)^h)$ states. Hybrid is better when $h=\omega(k^{4/5})$.

For comparison, the pebbling algorithm uses theta of k plus one to the h states.
It beats the pebbling algorithm when h grows faster than k to the four over five.
So, how does the hybrid algorithm work? Well, it's based on an encoding that combines the one-hot and binary encodings.

Hybrid algorithm

The Hybrid encoding is broken into $\frac{k}{2^a-1}$ *blocks* that are a bits long.

Hybrid algorithm

The Hybrid encoding is broken into $\frac{k}{2^a-1}$ *blocks* that are a bits long.

The hybrid encoding is broken up into a number of blocks that are a bits long, where a is some parameter. It's easiest to show this with an example.

Hybrid algorithm

The Hybrid encoding is broken into $\frac{k}{2^a-1}$ *blocks* that are a bits long.
For example, with $k = 9, a = 2$:

x	block 1	block 2	block 3	full encoding
1	01	00	00	010000
2	10	00	00	100000
3	11	00	00	110000
4	00	01	00	000100
5	00	10	00	001000
6	00	11	00	001100
7	00	00	01	000001
8	00	00	10	000010
9	00	00	11	000011

Hybrid algorithm

The Hybrid encoding is broken into $\frac{k}{2^a-1}$ *blocks* that are a bits long.
For example, with $k = 9, a = 2$:

x	block 1	block 2	block 3	full encoding
1	01	00	00	010000
2	10	00	00	100000
3	11	00	00	110000
4	00	01	00	000100
5	00	10	00	001000
6	00	11	00	001100
7	00	00	01	000001
8	00	00	10	000010
9	00	00	11	000011

When k is nine and a is two, here are the encodings of one through nine.
The encodings of 1, 2 and 3 use the first block, and leave the others at zero.
The encodings of four, five and six use only the second block. Seven, eight and nine use the last block.
Now, once we have the encoding, the next step is to see how to compute the encoding of a node's value based on the children's values.

Hybrid algorithm

The Hybrid encoding is broken into $\frac{k}{2^a-1}$ *blocks* that are a bits long.
For example, with $k = 9, a = 2$:

x	block 1	block 2	block 3	full encoding
1	01	00	00	010000
2	10	00	00	100000
3	11	00	00	110000
4	00	01	00	000100
5	00	10	00	001000
6	00	11	00	001100
7	00	00	01	000001
8	00	00	10	000010
9	00	00	11	000011

Each bit of $\text{Hybrid}_a(v)$ is a degree-$2a$ polynomial in $\text{Hybrid}_a(\ell)$ and $\text{Hybrid}_a(r)$.

Hybrid algorithm

The Hybrid encoding is broken into $\frac{k}{2^a-1}$ *blocks* that are a bits long.
For example, with $k = 9, a = 2$:

x	block 1	block 2	block 3	full encoding
1	01	00	00	010000
2	10	00	00	100000
3	11	00	00	110000
4	00	01	00	000100
5	00	10	00	001000
6	00	11	00	001100
7	00	00	01	000001
8	00	00	10	000010
9	00	00	11	000011

Each bit of $\text{Hybrid}_a(v)$ is a degree-$2a$ polynomial in $\text{Hybrid}_a(\ell)$ and $\text{Hybrid}_a(r)$.

We can compute the hybrid encoding of a node using a degree-2a polynomial applied to the encodings of its children.

Hybrid algorithm

The Hybrid encoding is broken into $\frac{k}{2^a-1}$ *blocks* that are a bits long.
For example, with $k = 9, a = 2$:

x	block 1	block 2	block 3	full encoding
1	01	00	00	010000
2	10	00	00	100000
3	11	00	00	110000
4	00	01	00	000100
5	00	10	00	001000
6	00	11	00	001100
7	00	00	01	000001
8	00	00	10	000010
9	00	00	11	000011

Each bit of $\text{Hybrid}_a(v)$ is a degree-$2a$ polynomial in $\text{Hybrid}_a(\ell)$ and $\text{Hybrid}_a(r)$.
Using this, we can build an algorithm that uses 3 registers with $\frac{ka}{2^a-1}$ bits each and makes $2^{\Theta(a)}$ recursive calls at each level, for a total of $2^{\Theta(ah)}k^{\Theta(1)}$ layers.

Hybrid algorithm

The Hybrid encoding is broken into $\frac{k}{2^a-1}$ *blocks* that are a bits long.
For example, with $k = 9, a = 2$:

x	block 1	block 2	block 3	full encoding
1	01	00	00	010000
2	10	00	00	100000
3	11	00	00	110000
4	00	01	00	000100
5	00	10	00	001000
6	00	11	00	001100
7	00	00	01	000001
8	00	00	10	000010
9	00	00	11	000011

Each bit of $\text{Hybrid}_a(v)$ is a degree-$2a$ polynomial in $\text{Hybrid}_a(\ell)$ and $\text{Hybrid}_a(r)$.
Using this, we can build an algorithm that uses 3 registers with $\frac{ka}{2^a-1}$ bits each and makes $2^{\Theta(a)}$ recursive calls at each level, for a total of $2^{\Theta(ah)}k^{\Theta(1)}$ layers.

We can follow the same pattern we used for the one-hot and binary algorithms to turn this encoding into another algorithm. When everything's put together, the algorithm uses three vector registers with k a over two to the a minus one bits each. It makes two to the theta a recursive calls and does k to a constant power work, for a total length of two to the theta of a h times k to a constant.
So, that concludes the algorithms I wanted to present.

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.)

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 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.