An acceleration and simplification of Ramachandran and Schild's algorithm for stabilizing sandpiles on trees: the round sequence Ramachandran-Schild algorithm

I want to make it clear that:

> **This note is based on the ideas of a preprint of Akshay Ramachandran and Aaron Schild, 'Sandpile prediction on a tree in near linear time,' which I saw as an anonymous reviewer.**

However, I think it is sufficiently novel to be worth writing.

First, let's talk about the original algorithm. Let the *critical component* of a vertex be the connected component of critical (or supercritical) vertices. If a vertex is supercritical, we can *execute a round* at it by firing everything in the critical component of that vertex exactly once. All of these firings will be legal if we carry them out in increasing order of distance from the supercritical vertex. The net effect is to move one chip across each boundary edge of the critical component from the critical vertex to the subcritical one.

We can simulate the sandpile by starting with an empty pile and dropping chips one by one on the vertices. Every time that a vertex becomes supercritical during this process, we execute rounds at that vertex until it is critical again.

Ramachandran and Schild try to simulate the sandpile this way, with the optimization that they are able to shortcut all the rounds until the next occurrence of a certain unusual event with one single fast computation. The unusual event occurs $n \log n$ times in the worst case. Each time it happens, at worst $\log^4 n$ work is done, using an interesting data structure.

The new idea that I present here is the idea of the 'round sequence,' which simplifies the problem substantially. It turns out that you can avoid three log powers by forgetting about simulation. Instead, you write down sequences that summarize the behaviour of subtrees, and combine them with an easy merging process to get the sequence of the whole tree.

The round sequence of a sandpile is the same as the round sequence of its stabilization. So, once you have found the round sequence, you can work backwards to find out how many chips are on each vertex after stabilization. —At least, that is the idea. It is not quite that nice. You cannot recover the configuration from the round sequence in general (see the end of Section 1.4). So it is necessary to introduce the more complicated idea of a 'labeled round sequence,' and that spoils things a little bit since it is no longer invariant under stabilization. But there is still enough information to figure out the stabilized configuration.

This algorithm has the peculiar property that it is faster when there are more chips, because the round sequences are shorter. So it is hard to beat in terms of efficiency per chip, since it starts with a large fixed cost and then each extra chip takes *negative* time.

Hannah Cairns, May 25, 2016 (ahc238@cornell.edu)

Contents

1. Rooted sandpiles.

1.1. **The round sequence.**

Let a *rooted sandpile*, or r-pile, be a sandpile on a tree with one extra virtual edge going out of the root to an imaginary ancestor which is always critical. Whenever the root sends a chip to the ancestor, it immediately fires and returns the chip to the root.

In a standard sandpile, vertices become unstable when they have a number of chips equal to their degree. In an r-pile, all non-root vertices have threshold equal to their degree, but the root has one extra imaginary edge, so unlike all the other vertices, its threshold is its degree plus one. So, all vertices in the r-pile are critical at one chip per child.

If the r-pile has v vertices, then it takes $v-1$ chips to make every vertex critical; then we say it is *full*. If it has no supercritical vertices, we say it is *stable*.

If the root is critical, we can *execute a round* by moving a chip across every boundary edge of the critical component of the root. The imaginary ancestor is always critical, so the virtual edge is never a boundary edge. So, this process only moves chips around inside the r-tree; it does not change the overall number of chips in it.

If the root is subcritical, we can add a chip to it. For a stable r-pile, we can always do exactly one of those two things: execute a round or add a chip. If we repeatedly do whichever we can do, writing down the pattern of 'round' and 'chip,' we could reasonably expect it to tell us something about the behaviour of the r-pile.

We can add at most $v-1$ chips before the pile is full, so 'chip' appears finitely many times and there must therefore be an infinite tail of 'round's. But if the pile is not full, then when we execute a round, the distance from the root to the closest subcritical vertex decreases by one, and if we continue executing rounds the root will eventually become subcritical. In other words, if at some point the pile is not full, there will eventually be another 'chip.' So once we hit the infinite tail, the pile must be full.

Suppose S is a stable r-pile with v vertices that already has n chips in it. Then we define the *round sequence* as the product of the following algorithm:

```
1: function STABLE ROUND SEQUENCE(S)
2:     $round\text{-}number \leftarrow 0$; $roundseq \leftarrow ()$
3:     while $S$ is not full do
4:         if the root is critical then
```

```
 5:             move one chip across every boundary edge of the critical component
 6:             add one to *round-number*;
 7:         else add a chip to the root and append *round-number* to *roundseq*.
 8:         end if
 9:     end while
10:     return *roundseq*
11: end function
```

We execute rounds and add chips until the pile is full. It does eventually become full by the reasoning above. Every time we execute a round, we add one to the round number; every time we add a chip, we write down the current round number.

If S is not stable, we *stabilize* it by treating it like a standard sandpile in which the imaginary ancestor is a real vertex which is a sink. That is, when the non-root vertices fire, one chip lands on each neighbour like normal, and when the root fires, one chip lands on each child and one chip is lost to the ancestor.

The round sequence of S is defined as the round sequence of its stabilization:

```
1: function ROUND SEQUENCE($S$)
2:     stabilize $S$ as if the imaginary ancestor was a sink
3:     *ancestor-flow* $\leftarrow$ the number of chips that fell into the sink
4:     return STABLE ROUND SEQUENCE($S$), *ancestor-flow*
5: end function
```

The *ancestor flow* of S is the number of chips that fall in the sink while we stabilize.

Let S be the r-pile on the path $\bullet - \circ - \circ - \circ$, where $\bullet$ is the root and there are no chips to start with. The thresholds are $2-2-2-1$, and so if we call ROUND SEQUENCE, we get the sequence of rounds in Table 1.1.

Chip configuration	*root*	*action*	*round*	*roundseq*
$0-0-0-0$	subcritical	chip	0	()
$1-0-0-0$	critical	round	0	(0)
$0-1-0-0$	subcritical	chip	1	(0)
$1-1-0-0$	critical	round	1	(0, 1)
$1-0-1-0$	critical	round	2	(0, 1)
$0-1-1-0$	subcritical	chip	3	(0, 1, 3)
$1-1-1-0$	**tree full**			

TABLE 1. The output of the algorithm for $\bullet - \circ - \circ - \circ$.

1.2. Building up the round sequence.

Our first goal is to understand how to find the round sequence of an r-pile recursively. First, if we know the round sequence of S, we can find the round sequence of $\bullet - S$.

Lemma 1. *Suppose S is a stable r-pile with round sequence $r_1, \dots, r_{v-1}$. Add a new empty root vertex with the root of S as its child. Let this new pile be S'.*

Then the round sequence of this new r-pile is $(0, r_1+1, r_2+2, \dots, r_{v-1}+v-1)$.

Proof. The root starts subcritical, so we have to add one chip to make it critical. This accounts for the initial zero. It has one child, so after we have added one chip, it is critical.

After that, it takes r_1 rounds for the child to become subcritical, by the definition of the round sequence for the child. It takes one more round to move the chip on the root to the subcritical child. After that extra round, the root is subcritical. We add a chip to it, making it critical again, and append the current round number to the sequence: that is r_1+1.

Then it takes another r_2-r_1 rounds until the child is subcritical for the second time, plus another extra round to move the chip on the root to the child. The root is subcritical for the third time, and we append $r_1+1+(r_2-r_1)+1=r_2+2$ to the round sequence.

Every chip that is moved from the root to the child introduces a delay of one more round, so that when the child has been subcritical n times, there is a delay of n rounds. This proves the result. □

Repeated application of this lemma tells us the round sequences for all empty paths:

$$\begin{array}{ll}
\bullet & () \\
\bullet - \circ & (0) \\
\bullet - \circ - \circ & (0,1) \\
\bullet - \circ - \circ - \circ & (0,1,3) \\
\vdots &
\end{array}$$

And in general, the round sequence of a path with v vertices will be $(r_1,\dots,r_{v-1})$ where $r_i=i(i-1)/2$. This is easy to see inductively.

Suppose we have a collection of stable r-piles $S_1,\dots,S_n$. If the roots have $d_1,\dots,d_n$ children and have $c_1,\dots,c_n$ chips, we can combine the roots to make a single big r-pile whose root has degree $\sum d_j$ and which has $\sum c_j$ chips on it. The combined root is critical if $\sum d_j=\sum c_j$, and that will happen if and only if $c_j=d_j$ for each root.

Lemma 2. *Suppose S_1, ..., S_n are stable piles with roots that have degrees $d_1,\dots,d_n$ and that have $c_1,\dots,c_n$ chips on them. Make a new pile S' by combining all of the roots. The round sequence of S' is the sorted concatenation of the individual round sequences.*

Proof. Imagine that we run all of the individual r-piles in parallel. If at some point all the roots are critical, we execute a round on each sandpile and increase the round number by one. If they are not all subcritical, we add chips to the subcritical ones to make them critical again, and append the current round number to the round sequence once per added chip. The round sequence that we get is the sorted concatenation of the individual round sequences.

The threshold of the combined root is the same as the sum of the thresholds of the individual roots, and it is not difficult to see that the number of chips on the combined root is always the sum of the chips on the individual roots. So the round sequences of the parallel sandpile and the combined sandpile are the same. □

Lemma 3. *The round sequence of a single vertex is* $()$.

Proof. The vertex has no children, so it is full when it has zero chips on it. □

With these three lemmas, we can find the round sequence of any empty r-pile recursively. The round sequence of a leaf is (). Find the round sequences of all the r-piles on the children, apply the map in Lemma 1, concatenate, and sort.

1.3. **Adding a chip.**

Lemma 4. *Suppose S is an r-pile with some round sequence $r_1,\dots,r_{v-1}$. Add a chip to the root. The round sequence of the resulting sandpile is*

$$\begin{array}{ll} r_2,\dots,r_{v-1} & \textit{if } r_1 = 0 \\ r_1-1,\dots,r_{v-1}-1 & \textit{if } r_1 \neq 0 \textit{ or the sequence is empty} \end{array}$$

and the ancestor flow increases by 0 in the first case and 1 in the second case.

Proof. We must find the stabilization of the r-pile with the chip added. The abelian property of the sandpile tells us that we can stabilize, add the chip, and then re-stabilize.

If $r_1 = 0$, then the root is subcritical after the first stabilization, and so when we generate the round sequence for the original r-pile, the first step is to add a chip. If we add a chip beforehand, we will just skip the first loop of the round sequence algorithm, so one of the zeroes at the beginning is deleted.

If $r_1 \neq 0$, then the root of S is critical after stabilization. Adding one more chip after that will make it supercritical again, and so we have to re-stabilize. We can do that by executing one round with the imaginary ancestor behaving as a sink: i.e., we execute one normal round and then take the extra chip off the root and put it on the ancestor.

The overall effect is that we put a chip on the ancestor and then we execute a 'free' round before the stable part of the algorithm starts, so again we effectively skip the first loop. That lowers all the round numbers by one and increases the ancestor flow by one. □

Corollary. Take an r-pile and add c chips to the root, where $c \leq r_1$. The round sequence of the resulting sandpile is $r_1 - c,\dots,r_{v-1}-c$, and the ancestor flow increases by c.

1.4. The round sequence of an arbitrary r-pile.

Let the map of sequences

$$(r_1, r_2, \dots, r_m) \mapsto (r_1+1,\dots,r_m+m)$$

be called the *stretch map*. Lemma 1 tells us that we get the round sequence of $\bullet - S$ by stretching the round sequence for S and adding a zero in front.

In Lemma 4 we have an operation that deletes the first entry of the round sequence if it is zero, or subtracts one from every entry if it isn't, and in the second case adds 1 to the ancestor flow. Let this be called the *chip-add operation* on a pile's round sequence.

Lemma 5. *Suppose $S_1,\dots,S_d$ are r-piles, not necessarily stable. Let their round sequences be $R_1,\dots,R_d$, and let their ancestor flows be $c_1,\dots,c_d$. Connect all of their roots to a new root with c chips on it to get a new sandpile S''. Let $C = c + c_1 + \cdots + c_d$.*

The round sequence of the new r-pile is produced by stretching all the child round sequences, concatenating and sorting them, and then, if $C \leq d$, adding $d - C$ zeroes to the beginning, or if $C > d$, applying the chip-add operation $C - d$ times.

Proof. First, suppose that that every child is stable, so that $c_1 = \cdots = c_d = 0$ and $C = c$. Then we get the combined graph by

- adding an extra empty root vertex to each S_j to get $\bullet - S_j$;
- identifying all the root vertices to get a combined root;
- and adding c chips to the combined root.

The previous three lemmas tell us that we get the round sequence by

- stretching each round sequence $R_1,\dots,R_d$ and adding a zero to the front;
- concatenating and sorting the round sequences;
- and then doing the chip-add operation c times.

In the first step, we add d zeroes, one to the front of each sequence. They all wind up at the beginning when we concatenate and sort. Finally, we use the chip-add operation c times. If $c \leq d$, the chip-add operation just deletes c of the zeroes we added. If $c > d$, then the first c applications delete all of the zeroes we added, and then we do the chip-add operation another $c - d$ times. So the lemma is true in the stable case.

Now, suppose that some of the children are unstable. Then we start by stabilizing. Firing unstable non-root vertices inside of the r-tree doesn't affect the ancestor flow, and it doesn't affect the final stable configuration by the abelian property of the sandpile, so we can partially stabilize the tree without affecting the round sequence or the ancestor flow.

Fire unstable vertices in each child until it is stable, while leaving the root alone. Once this is done, all the children will be in their stabilized configurations, and the j-th child will have dropped c_j chips on the root by the definition of the ancestor flow, so the combined root will have $C = c + c_1 + \cdots + c_d$ chips on it. Then we are in the stable case above. $\square$

Now we can find the round sequence of any r-pile, however many chips it starts with and whether or not it is stable. Not only that, but we will see later that we can carry out all these operations very efficiently, in only $O(n \log^2 n)$ time for the whole pile.

The round sequence of a pile is the round sequence of its stabilization. So once we know the round sequence, we would like to be able to take it apart again to get the stabilized configuration. Unfortunately, that doesn't work: the merging step is not reversible. For example, if the round sequence of the configuration on $\circ - \circ - \bullet - \circ - \circ - \circ$ is (1), the chip configuration could be 002110 with one chip missing from the left, or 012010 with one chip missing from the right. This motivates the introduction of a *labeled round sequence*: we stick labels on all the numbers, and (we will see that) that makes merging reversible.

1.5. **Labeled round sequences: plain labeling and stable labeling.**

Let a *labeled sequence* be a sequence of nonnegative numbers labeled by vertices. If entries have the same round number but different labels, we do not order them, so that, say, $(1_a, 1_b)$ would be considered the same labeled sequence as $(1_b, 1_a)$.

Let a *labeled round sequence* of an r-pile be a labeled sequence that is equal to the round sequence of that pile when you strip off the labels. We will define two flavours.

First, let the *plain labeling* of (the round sequence of) an r-pile S be the output of the PLAIN LABELING function in Figure 1 on the next page. We construct the round sequence recursively with Lemma 5. We label each zero with the vertex where we added it, and keep track of those labels as the entry moves through the algorithm. Recursively at every vertex, we compute the plain labelings of the children, stretch them with the stretch map, merge them, and add $C - d$ zeroes to the start or use CHIP-ADD OPERATION $d - C$ times. We also keep track of the ancestor flows. We are following the rules in Lemma 5, so the result is the round sequence with some labels on it.

For example, the plain labeling of the pile $\circ_a - \bullet_b - \circ_c - \circ_d - \circ_e$ is $(0_b, 0_b, 1_c, 3_d)$ if there are no chips on it. Now suppose we add chips to the root one by one, find the plain labeling, stabilize, and find the plain labeling of the stabilization. See Figure 2.

Compare the second and fourth columns. The numbers are the same, as guaranteed by Lemma 5, but some of the labels are different. In fact, whenever the new chip that we add on that line makes the root unstable, at least one of the labels changes.

Let the *stable labeling* of a pile S be the plain labeling of the stabilization of S. This is also a labeled round sequence, because an r-pile and its stabilization have the same round sequence. There is a simple relationship between the plain labeling and the stable labeling, and we will see it in Lemma 8.

```
function CHIP-ADD OPERATION($R$, $ancestor\text{-}flow$)
    if $R$ is not empty and the first entry of $R$ is zero then
        delete the first entry of $R$
    else
        subtract 1 from the round number of every entry of $R$
        $ancestor\text{-}flow \leftarrow ancestor\text{-}flow + 1$
    end if
    return $R$, $ancestor\text{-}flow$
end function

function PLAIN LABELING($S$)
    $a \leftarrow$ the root of $S$
    $d \leftarrow$ the degree of $a$
    $c \leftarrow$ the number of chips on $a$
    $ancestor\text{-}flow \leftarrow 0$
    $b_1, \dots, b_d \leftarrow$ the children of $a$
    for $i = 1 \dots d$ do
        $R_i, c_i \leftarrow$ PLAIN LABELING(subtree of $S$ rooted at $b_i$)
        stretch $R_i$ using the stretch map (that is, send $R_{ij} \mapsto R_{ij} + j$)
    end for
    $R \leftarrow$ sorted concatenation of $R_1, \dots, R_d$
        ▷ Remember, if entries have the same round number, their order doesn't matter.
    $C \leftarrow c + c_1 + \cdots c_d$
        ▷ The number of chips on the site, plus all the ancestor flows from the children.
    if $C \leq d$ then
        for $i = 1, \dots, C - d$ do
            $R \leftarrow (0_a) + R$
        end for
    else
        for $i = 1, \dots, d - C$ do
            $R, ancestor\text{-}flow \leftarrow$ CHIP-ADD OPERATION($R, ancestor\text{-}flow$)
        end for
    end if
    return $R, ancestor\text{-}flow$
end function
```

FIGURE 1. How to generate the plain labeling

Chip configuration	*Plain label*	*Stabilization*	*Plain label of the stabilization*	*Flow*
$0_a - 0_b - 0_c - 0_d - 0_e$	$(0_b, 0_b, 1_c, 3_d)$	$0_a - 0_b - 0_c - 0_d - 0_e$	$(0_b, 0_b, 1_c, 3_d)$	0
$0_a - 1_b - 0_c - 0_d - 0_e$	$(0_b, 1_c, 3_d)$	$0_a - 1_b - 0_c - 0_d - 0_e$	$(0_b, 1_c, 3_d)$	0
$0_a - 2_b - 0_c - 0_d - 0_e$	$(1_c, 3_d)$	$0_a - 2_b - 0_c - 0_d - 0_e$	$(1_c, 3_d)$	0
$0_a - 3_b - 0_c - 0_d - 0_e$	$(0_c, 2_d)$	$0_a - 1_b - 1_c - 0_d - 0_e$	$(0_{\boldsymbol{b}}, 2_d)$	1
$0_a - 4_b - 0_c - 0_d - 0_e$	(2_d)	$0_a - 2_b - 1_c - 0_d - 0_e$	(2_d)	1
$0_a - 5_b - 0_c - 0_d - 0_e$	(1_d)	$0_a - 2_b - 0_c - 1_d - 0_e$	$(1_{\boldsymbol{c}})$	2
$0_a - 6_b - 0_c - 0_d - 0_e$	(0_d)	$0_a - 1_b - 1_c - 1_d - 0_e$	$(0_{\boldsymbol{b}})$	3
$0_a - 7_b - 0_c - 0_d - 0_e$	$()$	$0_a - 2_b - 1_c - 1_d - 0_e$	$()$	3

FIGURE 2. The labels change as we stabilize.

Lemma 6. *Let S be a stable r-pile with some plain labeling $(r_j)_{x_j}$, where $r_j \geq 0$ and x_j are vertices in S. Suppose the root is critical. Execute a round to get a new pile S'. The plain labeling of S' is $(r_1-1)_{y_1}, (r_2-1)_{y_2}, \ldots$, where y_j is x_j or its direct ancestor.*

Proof. We prove it by induction on the size of the r-pile. It is certainly true for the r-pile with a single vertex. Suppose that the statement is true for every r-pile with fewer than v vertices. Let S be a stable r-pile with root a.

Executing a round on S means executing a round on every critical child, and moving one chip from the root onto every subcritical child.

Let B be the set of all direct children of a. If $b \in B$ is subcritical, its plain labeling starts with 0_b; the stretched labeling will start with 1_b. When we move the chip from the root onto the child, the first 0_b in the plain labeling is deleted, and the *indices* of all the other entries go down by one. Therefore the stretched labeling will lose its first 1_b and the *round numbers* of the other entries will go down by one.

If b is critical, we execute a round, which by the induction hypothesis lowers all the round numbers in the plain labeling by one, possibly moves some of the labels up the tree one step, and doesn't delete anything. The round numbers in the stretched labeling all go down by one in this case also, but nothing is deleted.

We get the plain labeling of the root by merging the stretched labelings of the children and adding one 0_a for each chip that it takes to make the root critical. So, putting what we said above together, executing a round at a will delete one 1_b per subcritical child, lower all of the other round numbers by one, and move some of their labels up the tree. Each subcritical child also takes one chip off of the root, which adds one 0_a to the start.

But a is a direct ancestor of b. The overall effect is to subtract one from every entry and move some of the labels up the tree, which proves the induction step.

□

Definition. Let L be a sequence of nonnegative integers labeled with vertices. We *produce* other sequences from L by moving labels of entries in L up the tree. This is a transitive operation: if we can produce L' from L and L'' from L', then we can produce L'' from L.

Definition. We will say that L is a *prelabeling* of an r-pile S if we can produce the stable labeling of S from L, i.e. we can produce the plain labeling of the stabilization of S from L.

Now we will see that the various operations do not disturb this property of prelabeling.

If all the numbers in L are positive, let $L-1$ be the sequence we get when we subtract one from the number of every entry in L without changing any of the labels.

Corollary of Lemma 6. Let S be a stable r-pile with critical root. Suppose L prelabels S. Execute a round on S, giving a new r-pile S'. Then $L-1$ prelabels S'.

Proof. Let the stable labeling of S be R. Subtract one from every round number in R to get a new labeled sequence $R-1$. Lemma 6 tells us that that is a prelabeling of S'.

It is easy to see that we can produce $R-1$ from $L-1$. By the transitivity of production, we can produce the stable labeling of S' from $L-1$. □

Lemma 7. *Suppose we add a chip to the root of a stable r-pile S and re-stabilize, getting a new r-pile S'. If L prelabels S, then* CHIP-ADD OPERATION(L) *prelabels S'.*

Proof. Suppose we have an r-pile S with root a which has degree d and c chips.

If the root is critical, $c = d$, then there are no zeroes at the start of the round sequence and CHIP-ADD OPERATION$(L) = L-1$. As we saw in the proof of Lemma 4, the configuration of S' is obtained by executing exactly one round on S. Now use the corollary of Lemma 6.

If the root is subcritical, $c < d$, then the round sequence of S starts with zero. Therefore, CHIP-ADD OPERATION deletes the first zero from L. And the plain labeling of S' is the plain labeling of S with the first zero deleted and none of the other labels changed, so CHIP-ADD OPERATION(L) is a prelabeling of S'. □

Finally, we come to the point of all this.

Lemma 8. *The plain labeling of an r-pile is a prelabeling of it.*

Proof. We prove this by induction on the size of the r-pile. We find the stable labeling of S by taking the stable labelings of the children, stretching and merging them, adding zeroes, and then doing the chip-add operation until the root is stable.

If we compare that to constructing the stable labeling of the stabilized pile S', we make mistakes in two places: first, we use the original sequences of the children instead of the stabilized ones, and second, we use CHIP-ADD OPERATION, which does not account for the fact that labels are moving up the tree.

The only effect of those mistakes is that the computed labels don't move up the tree when the real labels do, so we are done. □

1.6. Reconstructing the child round sequences from a prelabeling.

Suppose we have a prelabeling L of a stable r-pile S. Each label in the plain labeling of the stable pile, the *real label*, is somewhere between the label in the prelabeling and the root. That's enough information to reconstruct the children.

Suppose we have an entry r_x in the prelabeling L. If $r = 0$, the real label is the root. If $r \neq 0$, the real label isn't the root, so it must be in a subtree — and it has to be between x and the root, so it must be in the same subtree as x. So, if we have a prelabeling L of the root, we count the zeroes at the start, throw them out, separate the rest into subtrees, and unstretch them, and we will get prelabelings of the children.

The function below does all this, and it also tells us how many chips are on the root.

```
function SLOW UNMERGE($L$)
    chips-on-root ← the degree of the root
    while $L$ begins with a zero do
        delete the starting zero
        chips-on-root ← chips-on-root − 1
    end while
    for $b$ a direct child of the root do
        $L_b$ ← ()
        delete all entries with label $b$ or child of $b$ from $L$
        put the deleted entries in the sequence $L_b$
        unstretch $L_b$
    end for
    return chips-on-root and all $L_b$.
end function
```

We can now stabilize an r-pile. The strategy is as follows: use PLAIN LABELING recursively to construct the plain labeling. That is a prelabeling of the stable sandpile. Then use UNMERGE recursively on it to find out how many chips are on each vertex.

The algorithms above are not very efficient. Let's talk about that.

1.7. Efficient merging, efficient unmerging.

We want to build up our plain labeling with a recursive construction at each vertex, using a data structure which makes each operation fast. Suppose that we can do every operation in time $O(\log n)$. Then how quickly can we do the recursion?

The plain labeling of a tree with n children can at worst, if the pile is empty, have $n-1$ entries. So if we do the merging naively, we could be doing an amount of work at each vertex proportional to the number of children of that vertex. If the graph is a path, that's $\Omega(n^2)$.

But we aren't naive; we do it cleverly. The whole idea is based on the following lemma, which says that we can work fast as long as we aren't charged for the largest child.

Lemma 9. *Let T be a tree with n vertices. At each vertex with degree d and subtrees of size $n_1 \geq \cdots \geq n_d$, charge a cost of $C(n_2 + \cdots + n_d)$. The total cost is $\leq Cn \lg n$.*

Proof. We may set $C = 1$.

We go by induction on the size of the tree. Suppose that the assumption is true for the subtrees. We need to show that the cost for T is $\leq n \lg n$. With the assumption on subtrees, the total cost for all the subtrees plus the cost at the root is bounded above by

$$\begin{aligned}
n_1 \lg n_1 + n_2 \lg n_2 + \cdots + n_d \lg n_d + (n_2 + \cdots + n_d)& \\
&= n_1 \lg n_1 + n_2(\lg n_2 + 1) + \cdots + n_d(\lg n_d + 1) \\
&\leq n_1 \lg n_1 + n_2 \lg n + \cdots + n_d \lg n \\
&\leq (n_1 + \cdots + n_d) \lg n \\
&\leq n \lg n.
\end{aligned}$$

That is the induction step, and we are done.[1] □

How can we avoid paying for the largest child when we are merging? Simple: we use a balanced tree structure to store the labelings. At every vertex, we decide which tree is largest and insert all the elements of the smaller trees into that. This will take one balanced tree operation The trees are never bigger than n, so balanced tree operations will not cost us more than $O(\log n)$.

Suppose the sizes of the child trees are $n_1, \ldots, n_d$. The trees are never bigger than n, so each operation takes at most $O(\log n)$. Inserting the small trees in the big one will take $(n_2 + \cdots + n_d)O(\log n)$ time at each vertex, so by the lemma the whole tree takes $O(n \log^2 n)$ time.

What about when we are unmerging? We keep a linked list of all entries with a certain label at each vertex, pointing to nodes in the red-black tree, and when we unmerge at a vertex, we go through all the labels in all of the small subtrees, delete all the entries with those labels from the root tree, and add them one by one in small trees, one for each subtree. Then we take the leftover tree and use that for the largest subtree. Extracting the small trees from the big one again takes only $(n_2 + \cdots + n_d)O(\log n)$ time at each vertex.

1.8. **Efficient stretching.** Okay, but we have to add linear polynomials to the tree as if it were a sequence, too. How can we do that? Well, we can just put the coefficients a, b of a linear polynomial $ax + b$ at each node of the red-black tree, and apply those coefficients to the values as we go through the tree operations.

The proposed data structure is a red-black tree with nodes that look like this:

[1]The bound can be improved to $\frac{1}{2}Cn \lg n$. The cost estimate becomes $\frac{1}{2}x_1 \lg x_1 + x_2(\frac{1}{2} \lg x_2 + 2) + \cdots + x_d(\frac{1}{2} \lg x_d + 2)$. This is convex and therefore the maximum over the polytope $0 \leq x_1 \leq n$, $0 \leq x_2, \ldots, x_d \leq n/2$, $\sum x_j \leq n$ has to be achieved at an extreme point, i.e. either $x_1 = n$ or $x_1 = n/2$, $x_j = n/2$ for some index j. In the first case the value is $\frac{1}{2}n \lg n$ and in the second it is $\frac{1}{4}n \lg(n/2) + \frac{1}{4}n \lg(n/2) + \frac{1}{4}n = \frac{1}{2}n \lg n$.

```
struct LINEARRBTREE
    enum $red\text{-}or\text{-}black \in \{red, black\}$          ▷ Red-black tree colour
    LINEARRBTREE* $left\text{-}child$, $right\text{-}child$          ▷ Red-black tree pointers
    LINEARRBTREE* $up$          ▷ Pointer to node above this one
    int $nleft, nright$          ▷ Size of left and right subtrees
    int $value$          ▷ Value of current node
    vertex $label$          ▷ Label of current node
    int $a$, $b$          ▷ Children get $a + bn$
end struct
```

plus some way to keep track of the nodes labeled with a particular vertex. For example, an array of linked lists, one per vertex, that lists the nodes labeled with that vertex. That is a little elaborate, and we propose a slicker idea in the “implementation” part.

As in Section 1.5, there is no order on labels. Nodes with the same *value* but different *label* will compare equal. We need a way to delete a specific node from the tree given a pointer to it, and the *up* field is there to help us manage that.

When we do a tree operation, every time we look at a node, before doing anything with it, we apply the coefficients to *value* and to the child nodes if any, and then set them to zero for the current node:

```
function PUSH COEFFICIENTS($node$)
    $node.value \leftarrow node.value + node.a \times node.nleft + node.b$
    $node.left\text{-}child.a \leftarrow node.a$
    $node.left\text{-}child.b \leftarrow node.b$
    $node.right\text{-}child.a \leftarrow node.a$
    $node.right\text{-}child.b \leftarrow node.b + node.a \times (node.nleft + 1)$
    $node.a \leftarrow 0$
    $node.b \leftarrow 0$
end function
```

This adds a constant cost per node that a tree operation looks at. If we do this, then the tree operations proper don’t need to know about these coefficients at all, because every node that they look at has $a = b = 0$. They do need to know about *nleft* and *nright* and maintain them as the tree changes, but that can be done in constant time per operation.

We must make sure to add only polynomials that don’t disturb the order of the elements of the tree. Polynomials with nonnegative coefficients will obviously preserve the order, and so will constants. During the unmerging we have to unstretch some stretched sequences, which means adding $-n$, but the values in the sequence are stretched, so there is always a gap of at least 1 between successive elements of the sequence, and the resulting tree is still ordered. (Remember two nodes with the same label compare equal.)

We have given all the pieces, and now we put together the algorithm.

2. Implementation

2.1. The stabilization algorithm for the r-pile.

With this data structure, here is the whole algorithm for toppling an r-pile:

```
function FAST R-PILE TOPPLING($S$)
    for each leaf $x$ do
        $T_x \leftarrow$ empty LINEARRBTREE
    end for
    for each non-leaf node $x$ in reverse DFS order do
        $C \leftarrow \sum$ ancestor flows of children $+$ chips on $x$
        $y \leftarrow$ the child of $x$ with the largest subtree
        $T_x \leftarrow T_y$ destructively
        stretch $T_x$ (add $n$)
        for each child $z \neq y$ of $x$ do
            stretch $T_z$ (destructively)
            merge $T_z$ into $T_x$        ▷ This part only takes $O(n \log^2 n)$ time total by Lemma 9.
        end for
        if $C \geq d$ then
        ▷ We want to use the CHIP-ADD OPERATION $C - d$ times. This could cost a lot if we did it
    the obvious way and paid $O(C - d)$, but we can do this more quickly: we write down how many
    chips are left, and subtract $\min(\textit{chips-left}, T_x[0])$ from all the round numbers and the number of
    chips remaining. If we have run out of chips, we stop there; otherwise we delete an entry from
    the round sequence, subtract one more from chips-left, and continue.
            $\textit{chips-left} \leftarrow C - d$
            while $\textit{chips-left} > 0$ do
                subtract $\min(\textit{chips-left}, T_x[0])$ from $T_x$ and chips-left
                if $\textit{chips-left} > 0$ then
                    delete the first entry of $T_x$
                    $\textit{chips-left} \leftarrow \textit{chips-left} - 1$
                end if
            end while
        else
            add $d - C$ zeroes to the start of $T_x$
        ▷ We add at most $n - 1$ entries here, and we can't delete more entries than we add, so
    the total number of deletions is $n - 1$. Addition and deletion cost $O(\log n)$, and subtraction
    costs $O(1)$, so the total cost of this part of the loop is only $O(n \log n)$.
        end if
    end for
                                                  ▷ We have just figured out the plain labeling.
                                                             ▷ Now take it all apart again.
    for each node $x$ in DFS order do
        $c_x \leftarrow$ the degree of $x$ minus the number of zeroes at the start of $T_x$
        delete all the zeroes at the start of $T_x$
        $y \leftarrow$ the child of $x$ with the largest subtree
        for each child $z \neq y$ of $x$ do
            $T_z \leftarrow$ empty LINEARRBTREE
            for each descendant $z'$ of $z$ do
                pull all the entries labeled with $z'$ out of $T_x$ and add them to $T_z$
            end for
            unstretch $T_z$
        end for
        unstretch $T_x$
        $T_y \leftarrow T_x$ destructively
```

```
42:     end for
43:     return all of the $c_x$
44: end function
```

We do at most $n \lg n$ additions during the merge and extractions during the unmerge, and the other operations, namely stretching, unstretching, zero addition, and zero deletion, happen at most n times each. The operations cost us $O(\log n)$ each, so the total cost of the whole algorithm is only $O(n \log^2 n)$.

2.2. **Stabilizing the sandpile.** We think of the sandpile as an r-pile with strange behaviour at the root. Each vertex in an r-pile is critical when it has one chip per child. The sandpile is the same, except for the root. The threshold of the root of a sandpile is one chip lower: if a sandpile and an r-pile have the same number of chips on each vertex, then the root will be supercritical in the sandpile exactly when it is critical or supercritical in the r-pile.

So here is how we can use the algorithm above to topple the sandpile. First, take all the chips off the root. Run the merging algorithm as above up to the point where we have to deal with the chips on the root vertex on line 14, but then, instead of doing the chip-add operation, just add d zeroes to the sequence as if the root were empty.

The result is a prelabeling of the r-pile we get by stabilizing everything but the root, and putting aside any chip that does land on the root. To finish, we must take every chip that we put aside and add it to the root with the sandpile dynamics. That is, we add one chip to the *stable* root, and check if the root is now *supercritical* in the sandpile sense. If it is, we execute rounds until it isn't supercritical anymore. We repeat until we have run out of chips and we are stable, or the tree is full and we know we will never stabilize.

In terms of the r-pile dynamics, we add a chip to the *subcritical* root, and then if it is *critical* with the threshold for the r-pile, we execute rounds until it isn't critical anymore. The round sequence tells us exactly how to do this.

We delete the first zero of the labeling (adding a chip to the subcritical root), and then subtract 1 from the sequence until the next entry is zero (executing rounds until subcritical):

```
function CHIP-ADD OPERATION FOR THE SANDPILE ROOT(T)
    delete the first entry of $T$                              ▷ Add a chip.
    if $T$ is empty then
        The tree is full: the sandpile will never stabilize.
    end if
    subtract $T[0]$ from $T$          ▷ Execute rounds until the sandpile root is stable.
                                       ▷ That is, until the r-pile root is subcritical.
end function
```

The result is a prelabeling of the r-pile corresponding to the stabilized sandpile. Again, this takes one heap operation per deletion, and there are at most n deletions total over the whole algorithm, so this does not cost too much.

This is the last piece that we need. The whole algorithm is given in Figure 3.

```
function FAST SANDPILE TOPPLING($S$)
    for each leaf $x$ do
        $T_x \leftarrow$ empty LINEARRBTREE
    end for
    for each non-leaf node $x$ in reverse DFS order do
        $C \leftarrow \sum$ ancestor flows of children $+$ chips on $x$
        $y \leftarrow$ the child of $x$ with the largest subtree
        $T_x \leftarrow T_y$
        stretch $T_x$ (that is, add $n$ to the sequence; see Section 1.4)
        for each child $z \neq y$ of $x$ do
            stretch $T_z$ (in place)
            merge $T_z$ into $T_x$
        end for
        if $x$ is not the root then                    ▷ We deal with the root in a special way.
            if $C \geq d$ then
                $\textit{chips-left} \leftarrow C - d$
                while $\textit{chips-left} > 0$ do
                    subtract $\min(\textit{chips-left}, T_x[0])$ from $T_x$ and $\textit{chips-left}$
                    if $\textit{chips-left} > 0$ then
                        delete the first entry of $T_x$
                        $\textit{chips-left} \leftarrow \textit{chips-left} - 1$
                    end if
                end while
            else
                add $0_x$ to the start of $T_x$, $d - C$ times
            end if
        end if
    end for
    $C \leftarrow \sum$ ancestor flows $+$ chips on root
    add $0_r$ to the start of $T_r$, $d - C$ times            ▷ Start with nothing on the root.
    for $i = 1 \ldots C$ do
        CHIP-ADD OPERATION FOR THE SANDPILE ROOT($T_r$)
    end for
                                   ▷ Now we have a prelabeling for the stabilized sandpile!
                                                    ▷ We take it apart using the labels.
    for each node $x$ in DFS order do
        $c_x \leftarrow$ the degree of $x$ minus the number of zeroes at the start of $T_x$
        delete all the zeroes at the start of $T_x$
        $y \leftarrow$ the child of $x$ with the largest subtree
        for each child $z \neq y$ of $x$ do
            $T_z \leftarrow$ empty LINEARRBTREE
            for each descendant $z'$ of $z$ do
                pull all the entries labeled with $z'$ out of $T_x$ and add them to $T_z$
            end for
            unstretch $T_z$ (that is, subtract $n$ from the sequence)
        end for
        unstretch $T_x$
        $T_y \leftarrow T_x$ destructively
    end for
    return all of the $c_x$
end function
```

FIGURE 3. The stabilization algorithm.

3. The implementation that comes with this note

That's the whole idea, but the Python implementation that is bundled with this has a few differences and subtleties that I thought I had better explain.

Unique labels.

On lines 25 and 30, we add a certain number of zeroes, labeled with the vertex where we have added them; and we never add more than the child degree of that vertex. In the actual code, in line 210-212 of `btree`, we label the zeroes with the *children* of the vertex where we added the zero, so that each entry can have a unique label. This changes the extraction procedure slightly also (`EXTRACT_ALL_CHILDREN` in `fastsand`).

The data structure is a 2-4 B-tree.

The structure proposed here uses a red-black tree because it is easier to write the structure, but the Python code uses a 2-4 B-tree instead of a red-black tree because it is less annoying to implement. It keeps track of the tree sizes for every node, and the up pointers for every node and also every entry; there are some unit tests that check this, which can be run by importing `btreetest` and calling `btreetest.unit()`.

The `delete_object` function.

For `EXTRACT_ALL_CHILDREN` (which implements lines 40–42 above), we keep a big array of entries with a certain label (the `labelrec` object in the `sandpile_tree` objects), and we iterate over them to pull them out. Normally, if you have a reference to an entry in a tree, you can just look at the key and delete it by that key. But in this tree, there are two technical problems with that. First, the key in the entry is not correct. To get the real key of the entry, you have to add all the linear polynomials above it in the tree. Second, the key is not enough, because we may have a large number of entries with the same key but different labels, and they are not sorted by label. So we maintain a table to tell us which node an entry is in and to let us go up the tree to the root (the up hash in the `btree` objects) and we have a separate `delete_object` function to delete a particular object from a tree regardless of its contents.

Testing.

The `fastsand.test()` function generates random trees, puts a lot of chips on one random vertex, and compares the fast stabilization algorithm (`fastsand.stabilize`) with the result of a slow $O(n^3)$ stabilization algorithm.

The `stabilize` function takes sandpiles with n vertices as tuples of two arrays of n nonnegative numbers, $(\texttt{tree}, \texttt{chips})$. The nodes of the tree are labeled $0, \ldots, n-1$, and zero is always the root vertex. The zeroth entry of `tree` is ignored, and for every other entry $\mathtt{a}[i]$ is the parent of i. The `chips` array just says how many chips are on the vertices.

The return value is a tuple. If the resulting sandpile does not stabilize, the first term is `False`; otherwise it is the stabilized sandpile. The second one gives some operation statistics: number of entries merged, number of entries extracted, number of `FIRST` calls, number of `POP` calls. For example, to simulate a sandpile on a line of eight vertices with four chips on the third vertex, call

```
fastsand.stabilize([0, 0, 1, 2, 3, 4, 5, 6],
                   [0, 0, 0, 4, 0, 0, 0, 0])
```

and the result should be `([0, 1, 1, 0, 1, 1, 0, 0], [0, 0, 15, 4])`. There are no merge or extract operations because there are no small subtrees.