# Stacks of Coins: Solution

We can model the problem as a directed graph.

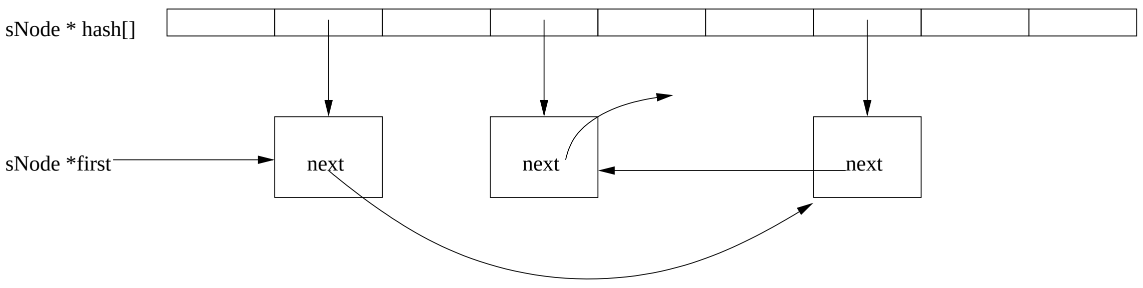
The situations before and after each move can be described as tuples consisting of the heights of the stacks: $(H_1, H_2, ..., H_S)$. These tuples are the vertices of our graph. A directed edge goes from vertex A to vertex B if and only if there is a valid move that transforms the stack tuple of A into the one of B. However, it is not necessary to represent this graph explicitly; only necessary vertices need to be generated in a dynamic way.

Now we can implement a breadth first search on the graph to find the shortest path from the starting situation to the flattened situation (where all stacks have the same height).

In this implementation, we first need a queue for the vertices of our graph. We use a linked list, where we append new vertices of the graph, whenever they occur.

To determine which vertices of the graph are new and which have already occurred, we need another datastructure. As there are a lot of tuples (number of coins$^{\text{number of stacks}}$) but only relatively few tuples that can be reached by valid moves, we use a hash. For test case 1 we need to visit about 1 million situations.

In my solution I have combined this hash and the list in one datastructure. It looks like this:



If we find a new vertex of our graph, we put it into the hash table and connect the next pointer of the last element of our list with the new vertex.

Finding and implementing this algorithm is quite simple. However, there is an improvement needed to cope with all test cases.

We can reduce the number of tuples that occur if we take into account that we do not need to differ between moves like $(8, 2, 2) \to (6, 4, 2)$ and $(8, 2, 2) \to (6, 2, 4)$. It is sufficient to store the heights of the stacks in descending order (ascending order is equally possible). That renders $(6, 4, 2)$ and $(6, 2, 4)$ equal (to $(6, 4, 2)$) and reduces the number of possible tuples. (In test case 1 from about 1 million to 10,000.)

Then, however, in order to reconstruct the path towards the flattened situation, we have to store the current permutation of stacks at each node in our data structure. This is done by adding indices to the tuple values. E.g., the (starting) position $(2, 8, 2)$ is stored as $(8_2, 2_1, 2_3)$. If we decide to move 2 coins from stack 2 to 3 (which are stored at positions 1 and 3 in the tuple) the situation first looks like this: $(6_2, 2_1, 4_3)$, but after sorting it is $(6_2, 4_3, 2_1)$. Further we store which moves we have done. So a solution for the starting situation $(2, 8, 2)$ looks like:

$$(8_2, 2_1, 2_3)_{\text{no last move}} \to (6_2, 4_3, 2_1)_{2\to 3} \to (4_2, 4_3, 4_1)_{2\to 1}$$

The corresponding output is:

```
2
2 3
2 1
```

This whole idea of pruning redundant stack permutations reduces search space and is fast enough for difficult test cases.

*Task and solution are proposed by Dominic Battré (dominic@battre.de).*