We are also asked to enumerate maximal sets of compatible splits. The straightforward reading of the problem statement suggests that this means maximal sets of (pairwise) compatible splits. However, one can naturally extend the definition of compatibility to include more than two splits at once: A set of n splits is compatible if there exists a tree that contains internal edges corresponding to each of these n splits. It seems reasonable to believe that these two ways to interpret "maximal sets of compatible splits" might be equivalent, however, we have not been able to prove that this is the case. For this reason, we cover each case individually:

Finding maximal sets of pairwise compatible splits: Since this is determined only by pairwise compatibility, one can build the adjacency matrix of this relation by comparing each pair of splits and storing whether they are compatible. Then, one can find all maximal sets of compatible splits with a naïve backtracking algorithm: Keep a set K of pairwise compatible splits (initialized to $\varnothing$), add splits that keep K internally compatible, and whenever no such split exists, print K and backtrack (note that we circumvent describing the actual backtracking procedure — essentially, the algorithm will have to be careful to try every path[1] exactly once).

Finding maximal sets of generally compatible splits: To give an algorithm for this, first we have to prove a couple statements about compatible sets:

Consider a set S of $m < n - 2$ compatible splits, and we will show that it is not maximal. By definition, there must be some tree T with n leaves and m internal edges such that these edges correspond to the splits in S. By counting, we have that T has $n + m + 1$ nodes (leaves, internal nodes, root), and thus $n + m$ edges. Notice that all leaves have degree 1, and let $d_1, \ldots, d_m$ denote the degrees of the internal nodes, with d_0 being the degree of the root. Then, by the handshaking lemma, $2(n + m) = n + \sum_{i=0,\ldots,m} d_i$, which we can rewrite as $n + 2m = d_0 + \sum_{i=1,\ldots,m} d_i$. Now, note that internal edges have degree at least 3, and the root at least 2 — for the sake of contradiction, assume that the degrees are *exactly* these values. Then, we obtain that $n + 2m = 2 + 3m \iff n - 2 = m$, a contradiction — so either the root has degree ≥ 3, or some internal edge has degree ≥ 4. Both of these scenarios mean that either the internal vertex of root can have an extra edge added, taking away 2 of its incident edges, while having the tree remain valid — this new tree would clearly contain all the same splits as the previous tree, plus a new one from the new internal edge. So, we obtain that:

- Maximal sets of compatible splits have $n - 2$ splits.
- The trees they correspond to are binary.

So a procedure to enumerate maximal splits could be:

- Enumerate all binary trees with n leaves, up to isomorphism.
- For each tree, enumerate all labelings, up to (tree) isomorphism.
- For each of these labelings (and the associated tree), print the set of compatible splits that they define.

Our implementation of this (in python) is available here, but a full description of this would be unreasonably long for this practical. We limit ourselves to explaining the two main algorithms. The first uses recursion — where **root** is some function that returns a single binary tree with a root and two nodes:

[1] "path" meaning nodes in K, in the order in which they were introduced in K.

1: ET(n): enumerate binary trees with n leaves up to isomorphism

```
Input: n
L = []
r = root()
if n = 1 then
    return [r]
end
for q_L in 1 : n − 1 do
    q_R = n − q_L
    if q_L ≠ q_R then
        for T_L, T_R in ET(q_L) × ET(q_R) do
            r_left = T_L
            r_right = T_R
            Append a copy of r to L
        end
    end
    else
        for T_L, T_R in ET(q_L) × ET(q_R), such that T_L comes ≤ T_R in ET(q_R) do
            r_left = T_L
            r_right = T_R
            Append a copy of r to L
        end
    end
end
return L
```

This description doesn't have any quirks: It just enumerates all trees by enumerating all pairs of subtrees at the first level, taking care that the amount of leaf nodes adds up. Moreover, when the amount of leaf nodes is equal in both subtrees, extra care is taken to ensure that every unordered pair is only added once, otherwise we would have two symmetric trees.

The second algorithm is likewise recursive, it takes a node v as input (from which it can access the entire tree below this v), and a set of labels to distribute among the leaf nodes below v. It should be called on the root node of the relevant tree, with $[n]$ as a set of labels.

2: EL($\mathbf{T}, \mathbf{l}$): enumerate labelings of T with labels l, up to isomorphism of T

Input: T, l
if *v is a leaf node* **then**
  return the labeling $v : l[0]$
end
Set v to be the root of T
Set T_L to be the tree below v_{left}
Set T_R to be the tree below v_{right}
Set q_L to the amount of leaf nodes below v_{left}
Set q_R to the amount of leaf nodes below v_{right}
$L = []$
if *the T_L and T_R are non-isomorphic* **then**
  for l_{left} *in* $\binom{l}{q_L} \setminus \{\varnothing, l\}$ **do**
    $l_R = l \setminus l_L$
    $L_R = EL(T_R, l_R)$
    $L_L = EL(T_R, l_R)$
    Extend L with every pair from $L_L \times L_R$
  end
end
else
  Choose p to be some element from l
  Remove p from l
  for l_{left} *in* $\binom{l}{q_L - 1} \setminus \{\varnothing, l\}$ **do**
    $l_R = l \setminus l_L$
    Add p to l_L
    $L_R = EL(T_R, l_R)$
    $L_L = EL(T_R, l_R)$
    Extend L with every pair from $L_L \times L_R$
  end
end
return L

This algorithm is analogous to the last, in that it recieves some parameter as an input (a set of labels), and it iterates over all the possible ways to share these labels among its left and right subtrees. For each possible way to share them, every pair of labelings of the left and right subtrees induces a labeling of the whole tree. Similarly to before, extra care has to be taken when the left and right subtrees are isomorphic — if this is the case, we force some element of the label set to always go to the left tree, so as to break symmetry.

Note as well that it is not obvious how to determine if T_L and T_R are isomorphic — but in practice, since we only look at trees yielded by ET, it is enough that ET flags a node if its left and right subtrees are isomorphic, so that this flag is used by EL later. In practice, it is trivial to determine whether the two subtrees are isomorphic while they are being generated, because they both come from $EL(n-1)$ — meaning that they are only isomorphic if they are equal.