# Solution for task ‘keys‘

## 1 Subtask 1

Look for connected components.

## 2 Subtask 2

We want to compute $p[i]$ for each $i$. A simple way is to use BFS and restart the BFS procedure whenever we pick up a new key.

## 3 Subtask 3

Again, we want to compute $p[i]$ for each $i$. For each key type $k$, store a ‘blocked‘ list of vertices that we are currently unable to reach, but we can reach if we collect key type $k$. Whenever we visit a new vertex $u$, we classify the edges $(u, v, c)$ into 3 cases:

- Case 1: $v$ has already been visited. In this case we do nothing.
- Case 2: $v$ is not visited and we have the connecting key $c$. In this case we simply put $v$ into the BFS queue.
- Case 3: $v$ is not visited and we do not have the connecting key $c$. In this case, we put $v$ into the ‘blocked‘ list.

Whenever we collect a new key type, simply clear the ‘blocked‘ list and put the relevant vertices into the queue.

This allows the BFS to run in $O(n + m + k)$ for every root node. Thus we can compute $p[i]$ for all $i$.

## 4 Subtask 4

Let $S_u$ represent the set of key types a player starting at $u$ can get. Suppose that there is a connector joining rooms $u$ and $v$ unlocked by key type $c$. We can then conclude the following:

- If $c \in S_u$, then $S_v \subseteq S_u$ (*)

This suggests the following:

Initially, $S_i = \{r[i]\}$ for all $i$. We start with a list of connectors which do not satisfy (*). If this list is nonempty, we take any connector and apply the operation $S_u \leftarrow S_u \cup S_v$ where necessary. Once this happens, all edges adjacent to $u$ are added to the list.

It may be the case where after the list is empty, $S_u$ may not be correct. For example, consider the input:

- 3 rooms with starting keys $r[0] = 0, r[1] = 1, r[2] = 2$
- 2 connectors, connecting rooms (0 and 1), and (1 and 2). Both connectors require key type 0 to unlock.

A player starting at room 0 is able to collect all 3 types of keys, however the algorithm will stop at $S_0 = \{0, 1\}, S_1 = \{1\}, S_2 = \{2\}$.

We put this issue aside temporarily and pretend that we have correctly calculated $S_u$ for all $u$, and explain later why this does not actually matter for the purposes of solving this problem.

If there is a connector $(u, v, c)$ and $c \in S_u$, then we may draw an edge $u \to v$. After identifying the strongly connected components, we simply need to calculate $p[i]$ for the tail components.

Back to the situation where $S_u$ may not be correct. Observe that every strongly connected component must have the same set $S_u$ for each $u$ in the component. This is because an edge $u \to v$ is drawn only when $S_v \subseteq S_u$. If a player $u$ obtains a key not in $S_u$, then this key must be obtained in some $v$ which is in a different component from $S_u$. However, when this happens, it means that the component containing $u$ is not a tail component. Since we only want the minimal $p[i]$, this does not matter.

Now let's analyze the time complexity of this algorithm. Observe that each edge is added to the list at most 60 times since an edge is only added when $S_u$ is changed. The complexity of this operation is therefore $O(m \cdot \max c[i])$ (assuming that the set union operation takes $O(1)$ time, which is acceptable since we only need to store 30 bits).

# 5 Full solution

First, assign every room a root. Initially, the root of every room is itself. A round consists of the following:

- Identify all the rooms which root is itself
- For each of this rooms, run the BFS algorithm described in subtask 3, terminating early if we are able to reach a room with a different root.
- Whenever an early termination occurs, we write down a list of pairs $(x, y)$ where 'a BFS search starting at $x$ reaches a room which root is $y$.

- At the end of the round, for all such pairs $(x, y)$, every room which root is $x$ (including $x$ itself), is now assigned a root of $y$. It does not matter how cycles are resolved, as long as each cycle contains a consistent root.

We can use the union find data structure to maintain the roots. The algorithm only needs to be ran for $\log_2(n)$ rounds. This can be proven by drawing out the merging tree - if in any iteration a room is not involved in any merge, then it will never be involved in any future merges.

Lastly, we finish by doing BFS on each of the roots.