Problem 1. A k-set-agreement object is a generalization of a consensus object in which processes could decide up to k different values. Formally, k-set-agreement is defined as follows. It has an operation \( \text{propose}(v) \) that returns (or we say decides) a value, which satisfies the following properties:

1. **Validity**: Decided values are proposed values.
2. **Agreement**: At most k different values could be decided.
3. **Termination**: Every correct process eventually decides a value.

A k-simultaneous-consensus object is another generalization of a consensus object in which processes could decide k values simultaneously. Formally, k-simultaneous consensus is defined as follows. It has an operation \( \text{propose}(v_1, \ldots, v_k) \) that returns (or we say decides) a pair \((\text{index}, \text{value})\) with \( \text{index} \in \{1, \ldots, k\} \), which satisfies the following properties:

1. **Validity**: If a process decides \((i, v)\), then some process proposed \((v_1, \ldots, v_k)\) with \( v_i = v \).
2. **Agreement**: If two processes decide \((i, v)\) and \((i', v')\) with \( i = i' \), then \( v = v' \).
3. **Termination**: Every correct process eventually decides a value.

Your task is to show that k-set-agreement and k-simultaneous-consensus are equivalent. That is, you have to show that one implements the other.

**Hint**: When implementing k-consensus using k-set-agreement, an algorithm that solves the problem is the following:

1. **function** `KSC.PROPOSE(v_1, \ldots, v_k)`
2. \( V_i \leftarrow [v_1, \ldots, v_k] \)
3. \( dV_i \leftarrow kSA.PROPOSE(V_i) \)
4. \( \text{REG}[i] \leftarrow dV_i \)
5. \( \text{snap}_i \leftarrow \text{REG.snapshot}() \)
6. \( c_i \leftarrow \text{number of distinct (non-\(\perp\)) vectors in snap}_i \)
7. \( d_i \leftarrow \text{minimum (non-\(\perp\)) vector in snap}_i \)
8. **return** \((c_i, d_i[c_i])\)
9. **end function**

Where \( \text{REG}[0, \ldots, n-1] \) is an array of single-writer multi-readers atomic registers initialized at \( \perp \). Processes write atomically a vector of values in their register (Line 4). \( \text{REG.snapshot}() \) returns an atomic snapshot of this array of registers. Consequently, \( \text{snap}_i[0, \ldots, n-1] \) is an array of vectors, possibly containing \( \perp \) values for some indices. We suppose that there is an order on the set of values that can be proposed, and we use the induced lexicographic order on vectors at Line 7.

Your task is then to (1) prove that the algorithm above implements a k-simultaneous consensus from k-set agreement objects and atomic registers; and (2) find an algorithm that implements a k-set agreement object using k-simultaneous consensus objects and atomic registers.
Solution

We will show that the \( k \)-set agreement problem and the \( k \)-consensus problem are equivalent. To do that we will show two wait-free constructions, one in each direction. Both constructions are independent of the number of processes.

From \( k \)-simultaneous-consensus to \( k \)-set agreement  

A pretty simple wait-free algorithm that builds a \( k \)-set agreement object (denoted KSA) on top of a \( k \)-consensus object (denoted KC) is described below. The invoking process \( p_i \) calls the underlying object \( KC \) with its input to the \( k \)-set agreement as input, and obtains a pair \( \{ci, di\} \). It then returns \( di \) as the decision value for its invocation of \( KSA.set\_propose_k(vi) \).

```c
KSA.set_propose_k(vi)
{
    \{ci, di\} = KSC.sc_propose_k([vi, vi, ..., vi]);
    return di;
}
```

**Proof.** The proof is straightforward. The termination and validity of the \( k \)-set agreement object follow directly from the code and the same properties of the underlying \( k \)-consensus object. The agreement property follows from the fact that at most \( k \) values can be decided from the \( k \) consensus instances of the \( k \)-consensus object.

From \( k \)-set agreement to \( k \)-simultaneous-consensus  

We prove that the presented algorithm (\( kSC.PROPOSE \)) satisfies validity, agreement, and termination.

The algorithm satisfies **termination** since we can implement a snapshot object in a wait-free manner and we assume that the \( k \)-set-agreement object satisfies termination.

Additionally, the algorithm satisfies **validity**. To see this, note that \( snap_i \) contains at most \( k \) vectors, where each vector is proposed by some process. Therefore, the vector \( d_i \) contains a proposed vector by some process and when a process returns \( (j, d_i[j]) \), it is the case that some process proposed a vector \( (d_i) \) with the \( j \)-th element being \( d_i[j] \).

Finally, the algorithm satisfies **agreement**. Assume by way of contradiction that this is not the case. This means that there exist two different processes \( p_a \) and \( p_b \) that decide \( (j, v) \) and \( (j, v') \) respectively and \( v \neq v' \). Process \( p_a \) performs the steps \( REG[a] \rightarrow snap_a \rightarrow c_a \rightarrow d_a \), while process \( p_b \) performs the steps \( REG[b] \rightarrow snap_b \rightarrow c_b \rightarrow d_b \). Since both \( p_a \) and \( p_b \) decide \( (j, v) \) and \( (j, v') \) we know that \( p_a \) reads \( c_a = j \) and \( p_b \) reads \( c_b = j \), but then \( d_a \) and \( d_b \) contain a different minimum vector. Without loss of generality assume that \( snap_a \) takes place before \( snap_b \), denoted by \( snap_a \rightarrow snap_b \). We now have the following two options:

- \( REG[a] \rightarrow REG[b] \rightarrow snap_a \rightarrow snap_b \), but then \( d_a = d_b \), a contradiction;
- \( REG[a] \rightarrow snap_a \rightarrow REG[b] \rightarrow snap_b \), but then \( c_a \neq c_b \), a contradiction.

In both cases, we reach a contradiction, and hence the algorithm satisfies agreement.