Problem 1:

Idea: Construct a directed graph \( G(V,A) \) from the given function \( f \) in the following manner. Let \( V = \{1,2,\ldots,n\} \). Let \( A \) contain each directed edge \((i,j)\) such that \( f(i) = j \). Since \( f \) is a (single-valued) function, the outdegree of each node in \( G \) is exactly 1. (Thus, \(|A| = n| \).

Suppose \( i \in V \) is a node such that indegree\((i) = 0\). Thus, there is no \( x \in S \) such that \( f(x) = i \). In such a case, \( i \) cannot be in the solution set \( X \) (since the range of the function \( f_X \) must \( X \) itself). Now, suppose the removal of node \( i \) and the outgoing edge from \( i \) produces another node \( k \) such that indegree\((k) = 0\). Then, since \( i \) is not in the solution set \( X \), node \( k \) cannot be in the solution set either. Thus, we can repeatedly remove from \( G \) nodes of indegree zero until all the remaining nodes have indegree of at least 1. These remaining nodes constitute the set \( X \). Since we removed only those nodes which cannot be in the solution, the remaining set of nodes is also the largest set satisfying the required properties. (This will be established formally.)

High-Level Description of the Algorithm: In the following description, we use a FIFO queue to obtain a running time of \( O(n) \).

1. Construct a directed graph \( G(V,A) \) from the given function \( f \) in the following manner. Let \( V = \{1,2,\ldots,n\} \). Let \( A \) contain each directed edge \((i,j)\) such that \( f(i) = j \), \( 1 \leq i \leq n \).

2. Compute the indegree of each node in \( V \).

3. Create a FIFO queue \( Q \) with each node \( i \) such that indegree\((i) = 0 \).

4. While \( Q \) is not empty do

   (a) Remove the first node \( i \) from \( Q \). Let \((i,j)\) be the outgoing edge from \( i \).

   (b) Delete node \( i \) from \( V \) and the edge \((i,j)\) from \( A \).

   (c) Set indegree\((j) = \) indegree\((j) - 1 \).

   (d) If indegree\((j) = 0 \), add \( j \) at the end of \( Q \).

5. Output the resulting set \( V \) as the solution set \( X \).

Proof of correctness: When the algorithm ends, each node in \( V \) has an indegree at least of 1. Thus, each value in \( X = V \) appears in the range of \( f_X \). The maximality of the set \( X \) is a consequence of the following lemma, which shows that none of the nodes eliminated from \( V \) by the above algorithm can be in the solution set \( X \).

Lemma 1: Let \( i_1, i_2, \ldots, i_r \) denote the nodes removed from \( V \) by the above algorithm in that order. For \( 1 \leq k \leq r \), node \( i_k \), cannot appear in the solution set \( X \).

Proof of Lemma 1: The proof is by induction on \( k \).
Basis: \( k = 1 \). The algorithm eliminated node \( i_1 \) because \( \text{indegree}(i_1) = 0 \). Thus, there is no \( x \in S \) such that \( f(x) = i_1 \). Hence, \( i_1 \) cannot appear in the range of the function \( f_X \). In other words, \( i_1 \) is not in the solution set \( X \).

Induction Hypothesis: Assume that for some \( k \geq 1 \), nodes \( i_1, i_2, \ldots, i_k \) are not in the solution set \( X \).

To prove: Node \( i_{k+1} \) is not in the solution set.

Proof: When the algorithm eliminated node \( i_{k+1} \), \( \text{indegree}(i_{k+1}) = 0 \). Thus, each value \( x \) such that \( f(x) = i_{k+1} \) has been eliminated before \( i_{k+1} \). By the inductive hypothesis, no such value \( x \) can be in the solution set \( X \). Hence, \( i_{k+1} \) cannot appear in the range of the function \( f_X \). In other words, \( i_{k+1} \) is not in the solution set \( X \), and this completes the inductive proof.

Running Time Analysis:

- As mentioned earlier, for the graph \( G(V, A) \), \( |V| = |A| = n \). Thus, Step 1 (constructing the graph \( G \)) can be done in \( O(n) \) time.
- By going through the adjacency list, computing the indegree of each node (Step 2) can be done in \( O(n) \) time.
- Since each insert operation in a FIFO queue can be done in \( O(1) \) time and \( Q \) has at most \( n \) nodes, creating the FIFO queue \( Q \) (Step 3) can be done in \( O(n) \) time.
- The loop in Step 4 iterates at most \( n \) times. In each iteration of the loop, each of the four steps can be carried out in \( O(1) \) time. Thus, the time for Step 4 is \( O(n) \).
- Since \( |V| \leq n \), Step 5 also runs in \( O(n) \) time.

Hence, the running time of the algorithm is \( O(n) \).

Problem 2:
Before proving the required result, we prove a lemma about any DFS spanning tree of a connected graph. This lemma uses the following notion of an ancestor in a tree. Given a tree \( T \) with root node \( r \) and two nodes \( a \) and \( b \) in \( T \), \( a \) is an ancestor of \( b \) if \( a \) lies in the path from \( b \) to \( r \).

Lemma 2: Suppose \( x \) and \( y \) are nodes in an undirected connected graph \( G(V, E) \) such that the edge \( \{x, y\} \) is in \( E \). Then, in any DFS spanning tree \( T \) of \( G \), \( x \) is an ancestor of \( y \) or \( y \) is an ancestor of \( x \).

Proof: Let \( d[x] \) and \( d[y] \) denote the discovery times of nodes \( x \) and \( y \) respectively when DFS is carried out on \( G \). We may assume without loss of generality that \( d[x] < d[y] \). Consider the time when the edge \( \{x, y\} \) is seen for the first time in DFS. There are two cases to consider.

Case 1: Edge \( \{x, y\} \) is seen for the first time from the adjacency list of \( x \). In this case, \( x \) is on top of stack and there are three possible colors for node \( y \).

Case 1.1: Color of \( y \) is white. In this case, edge \( \{x, y\} \) becomes a tree edge; thus, \( x \) is an ancestor of \( y \) in \( T \).

Case 1.2: Color of \( y \) is gray. In this case, node \( y \) is below \( x \) in the stack, implying that \( d[y] < d[x] \). This contradicts our assumption about discovery times. Hence, this case cannot arise.
Case 1.3: Color of $y$ is black. In this case, edge $\{x, y\}$ has already been seen from the adjacency list of $y$. This contradicts our assumption that edge $\{x, y\}$ is being seen for the first time. Hence, this case also cannot arise.

Thus, in Case 1, $x$ is an ancestor of $y$.

Case 2: Edge $\{x, y\}$ is seen for the first time from the adjacency list of $y$. In this case, $y$ is on top of stack. Again, there are three possible colors for node $x$.

Case 2.1: Color of $x$ is white. This case cannot happen since $d[x] < d[y]$.

Case 2.2: Color of $x$ is gray. In this case, $x$ is in the stack and there is a path from $x$ to $y$. Since there is always a path from $x$ to the root, $x$ is an ancestor of $y$.

Case 2.3: Color of $x$ is black. In this case, edge $\{x, y\}$ has already been seen from the adjacency list of $x$. This contradicts our assumption that edge $\{x, y\}$ is being seen for the first time. Hence, this case cannot arise.

Thus, in Case 2 also, $x$ is an ancestor of $y$, and this completes the proof of Lemma 2.

We are now ready to prove the required result.

**Theorem 3:** Let $G$ be a connected undirected graph. Suppose the same spanning tree $T$ is produced when we do a BFS or a DFS from a node $s$. Then, $G$ cannot contain any edge which is not in $T$.

**Proof:** We prove the result by contradiction. So, assume that $G$ has an edge $\{x, y\}$ which is not in $T$.

Consider $T$ as a BFS tree. Let us say that a node $v$ of $G$ is at level $i$ if its distance from $s$ in $T$ is equal to $i$. There are three possible cases depending on the levels of nodes $x$ and $y$.

Case 1: The levels of $x$ and $y$ are equal.

Consider $T$ as a DFS spanning tree of $G$. Since $x$ and $y$ are at the same level of $T$, $x$ is not an ancestor of $y$ and $y$ is not an ancestor of $x$ in $T$. However, since $\{x, y\}$ is an edge in $G$, this contradicts Lemma 2. Thus, in this case, $\{x, y\}$ cannot be in $G$.

Case 2: The levels of $x$ and $y$ differ by exactly 1.

Without loss of generality, let the level of $x$ be smaller than that of $y$. Once again, note that since $T$ does not have the edge $\{x, y\}$, $x$ is not an ancestor of $y$ and $y$ is not an ancestor of $x$ in $T$. However, since $\{x, y\}$ is an edge in $G$, this again contradicts Lemma 2. Thus, in this case, $\{x, y\}$ cannot be in $G$.

Case 3: The levels of $x$ and $y$ differ by 2 or more.

Without loss of generality, let $x$ have a smaller level number than $y$. Since the edge $\{x, y\}$ is in $G$, during the BFS of $G$, $y$ would have been assigned a level that is exactly one more than that of $x$. This contradicts the assumption that the levels of $x$ and $y$ differ by 2 or more. Thus, in this case also $\{x, y\}$ cannot be in $G$, and the theorem follows.

---

**Problem 3:**

**Idea:** We use another array $B[1..n]$. For $1 \leq i \leq n$, let $B[i] = \sum_{k=1}^{i} X[k]$; that is, $B[i]$ stores the sum of the elements in the subarray $X[1..i]$. The usefulness of storing these partial sums in $B$ is shown by the following lemma.

**Lemma 4:** There is a subarray of $X$ with sum equal to zero if and only if either $B[i] = 0$ for some index $i$ or $B[i] = B[j]$ for some pair of distinct indices $i$ and $j$. 3
Proof:

If part: If \( B[i] = 0 \) for some \( i \), it follows directly that the sum of the elements of the subarray \( X[1..i] \) is zero. So, suppose that there is a pair of indices \( i \) and \( j \), with \( i \neq j \), such that \( B[i] = B[j] \). We may assume without loss of generality that \( i < j \). Note that \( B[i] = \sum_{k=1}^{i} X[k] \) and \( B[j] = \sum_{k=1}^{j} X[k] \). Therefore, \( B[i] = B[j] \) implies that \( \sum_{k=i+1}^{j} X[k] = 0 \); in other words, the sum of the elements in the subarray \( X[i+1..j] \) is zero. (Since \( i < j \), the subarray \( X[i+1..j] \) has at least one element.)

Only If part: Suppose the sum of the elements in the subarray \( X[i..j] \), where \( i \leq j \), is zero. If \( i = 1 \), then it follows that \( B[j] = 0 \). So, assume that \( i \geq 2 \). Now, the given condition implies that the two elements \( B[i-1] = \sum_{k=1}^{i-1} X[k] \) and \( B[j] = \sum_{k=1}^{j} X[k] \) are equal. Since \( i \leq j \), the indices \( i-1 \) and \( j \) are distinct. The lemma follows.

In view of the above lemma, the algorithm needs to only compute the elements of the array \( B \) and check whether any of them is 0 or any pair of them are equal. The task of checking whether \( B \) contains two values that are equal can be done by sorting the array \( B \) and checking whether any pair of adjacent values are equal.

Steps of the Algorithm:

1. /* Compute the values of the elements of array \( B \). */
   
   Let \( B[1] = X[1] \) and for \( 2 \leq i \leq n \), let \( B[i] = B[i-1] + X[i] \).

2. If \( B[i] = 0 \) for some \( i \), then output “Yes” and stop.

3. Sort the array \( B \) using Mergesort.

4. By a simple linear scan of \( B \), check whether any pair of adjacent values \( B[i] \) and \( B[i+1] \) are equal for some \( i \), \( 1 \leq i \leq n-1 \). If so, output “Yes”; otherwise, output “No”.

Correctness of the Algorithm: This is a simple consequence of Lemma 4.

Running time Analysis: Step 1 runs in \( O(n) \) time since computing each element of \( B \) uses only \( O(1) \) time. Step 2 also runs in \( O(n) \) time. Step 3, which uses Mergesort, runs in \( O(n \log n) \) time. (Any sorting algorithm with a worst-case running time of \( O(n \log n) \) can be used in Step 3.) Step 4 runs in \( O(n) \) time since it involves just a linear scan of the array. So, the overall running time of the algorithm is \( O(n \log n) \).