1. (a) EXACTCYCLE $\in NP$: On input (G, k, c), where c is a list of vertices, verify that c contains exactly k vertices and that G contains every edge from one vertex in c to the next, and also from the last vertex back to the first one.

   This takes only polynomial time and it outputs True for some value of c iff G contains some cycle on exactly k vertices.

   EXACTCYCLE $\notin P$: HAMCYCLE $\leqslant_p$ EXACTCYCLE (where HAMCYCLE is the Hamiltonian Cycle, or "Rudrata Cycle", problem).

   On input $G = (V, E)$, output (G, n), where $n = |V|$.

   Clearly, (G, n) can be computed from G in polytime. Also, G contains a simple cycle on exactly n vertices iff G contains a Hamiltonian/Rudrata cycle, by definition.

   (b) SMALLCYCLE $\in P$:

```
for each $v \in V$:
    run BFS starting from $v$
    if BFS detects a cycle, traverse it to compute its length
    if the cycle contains $k$ or fewer vertices:
        return True
return False   # after trying every $v$
```

   This algorithm runs in polynomial time because it makes a linear number of calls to BFS (each one of which takes linear time), in addition to a linear number of cycle traversals (each one of which also takes linear time), in the worst-case.

   Also, if the algorithm returns True, the graph contains some cycle on at most k vertices (because the algorithm has found such a cycle explicitly).

   Finally, if the graph contains some cycle C on at most k vertices, then the algorithm returns True: when running BFS from one of the vertices on C, the algorithm will detect the existence of C, and the fact that C contains no more than k vertices (because BFS always finds paths, and cycles, that use as few edges as possible).

   Note that using DFS instead of BFS would not be guaranteed to work because DFS may not find every short cycle in the graph.

   (c) LARGECYCLE $\in NP$: On input (G, k, c), where c is a list of vertices, verify that c contains at least k vertices and that G contains every edge from one vertex in c to the next, and also from the last vertex back to the first one.

   This takes only polynomial time and it outputs True for some value of c iff G contains some cycle on at least k vertices.

   LARGECYCLE $\notin P$: HAMCYCLE $\leqslant_p$ EXACTCYCLE (where HAMCYCLE is the Hamiltonian Cycle, or "Rudrata Cycle", problem).

   On input $G = (V, E)$, output (G, n), where $n = |V|$.

   Clearly, (G, n) can be computed from G in polytime. Also, G contains a simple cycle on at least n vertices iff G contains a Hamiltonian/Rudrata cycle, by definition.

2. (a) The DIVERSITY optimization problem can be represented by the following Integer Program.

   Variables: $c_1, c_2, \ldots, c_n$ — each variable c_i represents whether or not to choose customer number i

   Objective function: *maximize* $c_1 + c_2 + \cdots + c_n$ — this represents the total number of customers chosen

   Constraints: $0 \leqslant c_i \leqslant 1$ for $i = 1, 2, \ldots, n$ — each customer is either chosen or not
   $\sum_{1 \leqslant i \leqslant n, T(i,j) > 0} c_i \leqslant 1$ for $j = 1, 2, \ldots, m$ — no two customers can have purchase j in common, for each j

Then every subset of customers C (no two of whom have a purchase in common) yields values $c_i = 1$ if customer i belongs to C; $c_i = 0$ otherwise. These values satisfy every constraint: in particular, for every product j, at most one customer who purchased product j belongs to C so $\sum_{1 \leqslant i \leqslant n, T(i,j)>0} c_i \leqslant 1$. This means that the maximum value of the objective function is at least as large as the maximum size of a subset of customers, no two of whom share a purchase.

Also, every feasible solution to the Integer Program yields a subset of customers C: pick the customers i with $c_i = 1$. No two customers in this subset share a purchase because $\sum_{1 \leqslant i \leqslant n, T(i,j)>0} c_i \leqslant 1$. So the maximum size of a subset of customers with no common purchase is at least as large as the maximum value of the objective function.

(b) The DIVERSITY decision problem is defined as follows:

Input: A table of purchases T (as in the question) with $n \geqslant 1$ customers and $m \geqslant 1$ products, and a positive integer k.

Question: Is there some subset of *at least* k customers, no two of whom have any purchase in common?

(c) DP $\in$ *NP*: On input (T, k, c), where c is a subset of customers, it takes polytime to verify that c contains at least k customers and that no two customers in c have a purchase in common.

DP is *NP*-hard: We show IS $\leqslant_p$ DIVERSITY (where IS is the INDEPENDENTSET problem).

On input (G, k) (where $G = (V, E)$ with $V = \{v_1, \ldots, v_n\}$ and $E = \{e_1, \ldots, e_m\}$), output (T, k) where T has "customers" $\{v_1, \ldots, v_n\}$ (*i.e.*, one customer for each vertex in G) and "products" $\{e_1, \ldots, e_m\}$ (*i.e.*, one product for each edge in G), and

$$T(v_i, e_j) = \begin{cases} 1 & \text{if } v_i \text{ is one endpoint of } e_j, \\ 0 & \text{otherwise.} \end{cases}$$

Clearly, (T, k) can be computed from (G, k) in polynomial time.
Also, if G contains some independent set I of size k (or more), then I is a subset of "customers" for which no two customers have a purchase in common: the only way two customers can have a common purchase is if they are endpoints of the same edge, and by definition, there is no edge of G between any two vertices in I.
Finally, if T contains some subset C of k (or more) customers, no two of whom share a purchase, then the vertices in C form an independent set in G: there cannot be any edge between two vertices of C, because that would translate to a "product" in common between two customers of C.

3. SERVERLOCATION $\in$ *NP*: Given $(G = (V, E), k, d, c)$, where $c \subseteq V$, it takes polytime to verify that size$(c) \leqslant k$, and that every vertex in G is within distance d of some element of c (by running BFS from each vertex in G).

SERVERLOCATION is *NP*-hard: We show VERTEXCOVER $\leqslant_p$ SERVERLOCATION.

Main Idea: Create G' by taking $G = (V, E)$ and adding an extra vertex for each edge $e \in E$, connected to both endpoints of e — effectively turning each edge of G into a triangle. Keep k the same and set $d = 1$.

This **almost** works, but not quite! There are two technical details that require special handling:

- If G contains isolated vertices, they may or may not belong to a vertex cover of size k, but they must be servers.
  We'll handle this by actually removing all isolated vertices from G'.
- If any of the new, extra vertices belong to a server set of size k in G', how do we know there is a corresponding vertex cover that contains only "old" vertices (those of G)?
  We'll handle this by showing that old vertices can be swapped for new ones in any solution — see below for the details.

Details:

On input (G, k), where $G = (V, E)$:

- Let $V = \{v_1, \ldots, v_n\}$ and $E = \{e_1, \ldots, e_m\}$. Furthermore, let $V_0 \subseteq V$ be the set of every isolated vertex of G, *i.e.*, the vertices of G that have no edge at all attached to them.
- Let $V' = V - V_0 \cup \{x_1, \ldots, x_m\}$, *i.e.*, remove all isolated vertices and add one new vertex for every edge in G.
- Let $E' = E \cup \{(x_i, v_i), (x_i, u_i) : e_i = (u_i, v_i) \in E\}$, *i.e.*, for each edge e_i of G, add edges from each endpoint of e_i to the new vertex x_i (turning e_i into a triangle in G').
- If $|V - V_0| < k$, then output $(G' = (V', E'), |V - V_0|, 1)$; else output $(G' = (V', E'), k, 1)$.

Clearly, this algorithm runs in polytime (as a function of n, m).

Moreover, if G contains fewer than k non-isolated vertices (those in $V - V_0$), then those vertices alone form a vertex cover in G and a server set of size $|V - V_0|$ in G'. Otherwise, if G contains a vertex cover C of size k, then every vertex of G' (including the new vertices x_i) is within distance 1 of a vertex in C — because C covers every edge of G and G' contains no isolated vertex. Starting with $S = C$ and replacing every isolated vertex in S by some non-isolated vertex outside S yields a server set of size k in G'.

Finally, if G' contains a server set of size $|V - V_0|$, then G contains fewer than k non-isolated vertices (in $V - V_0$) and, together with a sufficient number of isolated vertices, these form a vertex cover in G. Otherwise, if G' contains a server set S of size k, then there is some server set S' of size k such that S' contains no "new" vertex x_i: simply replace each x_i in S with u_i or v_i (one of the endpoints of edge e_i). If both u_i and v_i already belong to S', then replace x_i with any other vertex outside S'. But then, S' is a vertex cover in G: since every x_i is outside S', it must be connected to some vertex in S', which means every edge e_i of G has at least one endpoint in S'.

Alternate reduction: It was also possible to show that 3SAT $\leqslant_p$ SERVERLOCATION.

Main Idea: From A in 3CNF, create a graph G with one node C_j for each clause C_j in A and one triangle x_i — x'_i — $\neg x_i$ for each variable x_i of A. Then, connect each C_j to all three literals that appear in C_j, and set $d = 1$ and $k =$ number of variables of A.

- This can be done in polytime.
- Every satisfying assignment of values to the variables of A yields a server set that consists of the nodes that correspond to true literals.
- Every server set in G must include at least one node from each triangle (otherwise the node x'_i would neither be a server nor connected to one), so every server set of size k contains exactly one node from each triangle. If variables are set according to the nodes in the server set (picking values arbitrarily when x'_i is in the server set), then F will be satisfied because each clause is "covered".