1. (a) Informally, in the travelling salesman problem one is given a set of cities and the distances between them. The task is to find the shortest route that visits every city exactly once and returns to the starting city.

More formally, a Hamiltonian cycle in a graph is a path that visits every node exactly once and then returns to its starting node. The definition is the same for directed and undirected graphs, but in this context the default is that the graphs are not directed. If additionally the graph is weighted, the cost of a Hamiltonian path is the sum of the weights of the edges along the path. We now define the Travelling Salesman Problem as

$$\text{TSP} = \{ \langle G, k \rangle \mid \text{weighted graph } G \text{ has a Hamiltonian cycle with cost at most } k \}.$$  

(b) In the knapsack problem, we are given for \( n \) items \( i = 1, \ldots, n \) their “values” \( v_i \) and “weights” \( p_i \), and the “capacity of the knapsack” \( K \). These are all natural numbers. The task is to find a collection of items that fits into the knapsack and has maximal value.

More formally,

$$\text{KNAPSACK} = \{ \langle K, V, v_1, \ldots, v_n, p_1, \ldots, p_n \rangle \mid \text{for some } I \subseteq \{1, \ldots, n\} \text{ we have } \sum_{i \in I} v_i \geq V \text{ ja } \sum_{i \in I} p_i \leq K \}.$$  

Although the details of encoding the input are usually not important, here we must take care to encode the numbers using decimal, binary or similar system where the length of the encoding for natural number \( x \) is \( O(\log x) \). Thus, if all the number in an instance of the knapsack problem are in range \( \{1, \ldots, m\} \), the size of the input is \( O(n \log m) \). This is significant, because actually the problem does have an algorithm that runs in time \( O(n^k m^k) \) (where \( k \) is roughly 2; this is a nice exercise in an algorithmic technique called dynamic programming). However, this time complexity is not polynomial in the size of the input, which would be required for the problem to be in \( P \).

(c) In the set covering problem we have a basic set \( X \) and a collection of subsets \( A_1, \ldots, A_n \). The task is to cover the basic set with the least possible number of subsets. More formally

$$\text{SETCOVER} = \{ \langle X, k, A_1, \ldots, A_n \rangle \mid \text{for some } I \subseteq \{1, \ldots, n\} \text{ we have } |I| \leq k \text{ and } \bigcup_{i \in I} A_i = X \}.$$  

2. For a graph \( G = (V, E) \), define its complement graph as \( \overline{G} = (V, \overline{E}) \). The definitions directly imply that a set of nodes \( I \subseteq V \) is independent in \( G \), if and only if it is a clique in the complement graph \( \overline{G} \).

Therefore, for all \( k \) we have \( \langle G, k \rangle \in \text{INDEPENDENTSET} \), if and only if \( \langle \overline{G}, k \rangle \in \text{CLIQUE} \). Hence the independent set problem can be solved as follows:

(a) Given the input \( \langle G, k \rangle \), construct the complement graph \( \overline{G} \).

(b) Further, construct the encoding \( w = \langle \overline{G}, k \rangle \).

(c) If \( w \in \text{CLIQUE} \), then accept; else reject.

Clearly steps (a) and (b) can be implemented in polynomial time (assuming any reasonable encoding for graphs). Hence, if we could solve the clique problem in step (c) in polynomial time, then we could also solve the independent set problem in polynomial time.

For the reverse direction, notice that \( \overline{G} = G \). Therefore the clique problem can be solved as follows:

(a) Given input \( \langle G, k \rangle \), construct \( \overline{G} \).
(b) Further, construct \( w = \langle G', k \rangle \).

(c) If \( w \in \text{INDEPENDENTSET} \), then accept; else reject.

Again if \( \text{INDEPENDENTSET} \in \text{P} \), then this can be made to run in polynomial time.

We can write the above solution more concisely by noticing that the function \( f: \langle G, k \rangle \mapsto \langle G', k \rangle \) can clearly be computed in polynomial time. Hence, based on the preceding remarks, it is a polynomial time reduction of \( \text{INDEPENDENTSET} \) to \( \text{CLIQUE} \). Thus \( \text{INDEPENDENTSET} \leq_{\text{P}} \text{CLIQUE} \), so by Theorem 7.31 in the textbook we know that if \( \text{CLIQUE} \in \text{P} \), then \( \text{INDEPENDENTSET} \in \text{P} \). By a similar reasoning, \( f \) is also a polynomial time reduction of \( \text{CLIQUE} \) to \( \text{INDEPENDENTSET} \). (This is unusual; usually of course the same \( f \) does not work as a reduction to both directions.) Again Theorem 7.31 shows us that if \( \text{INDEPENDENTSET} \in \text{P} \), then \( \text{CLIQUE} \in \text{P} \).

3. (a) Assume that we can solve the decision version of the clique problem in polynomial time. That is, some algorithm solves it and runs in time \( O(p(n)) \) for graphs with \( n \) nodes, where \( p \) is a polynomial. The optimisation version can be solved as follows:

   i. Initialise \( k \leftarrow n \) where \( n = |V| \).
   ii. If \( \langle G, k \rangle \in \text{CLIQUE} \), print \( k \) and halt.
   iii. Set \( k \leftarrow k - 1 \) and return to previous step.

Notice that by definition, the empty set is a clique in any graph, so the algorithm will print zero, unless it otherwise halts before \( k \) reaches 0. The value printed by the algorithm is the size of the largest clique, since obviously this cannot be larger than \( n \). The time complexity is \( O(np(n)) \), which is polynomial.

(b) Assume that we have a function \( \text{OPTCLIQUE}(G) \) that returns the size of the largest clique in graph \( G \) and runs in polynomial time \( O(q(n)) \), where \( n \) is the number of nodes. The search problem can be solved as follows:

   i. Initialise \( G' = G \). Let \( k = \text{OPTCLIQUE}(G) \).
   ii. Repeat for all \( v \in V \) (in some arbitrary order):
      
      A. Construct \( G'' \) by removing from \( G' \) the node \( v \) and any edges adjacent to \( v \).
      
      B. If \( \text{OPTCLIQUE}(G'') = k \), then let \( G' \leftarrow G'' \).
   iii. Print the set of all the nodes remaining in \( G' \).

The time complexity is \( O(nq(n)) \), which is polynomial. The algorithm maintains the invariant that \( G' \) contains a \( k \)-clique. The same set of nodes is of course a clique also in the original graph \( G \). Therefore the set of nodes printed by the algorithm will always contain a clique of maximum size.

We also need to show, that the algorithm does not print any extra nodes. Let \( U \) be the set of nodes printed by the algorithm and \( I \subseteq U \) some \( k \)-clique. By the preceding remarks, such a clique always exists. If \( v \notin I \), then when the algorithm inspects \( v \) all the nodes of \( I \) are still in \( G' \), so \( v \) will be removed. Hence, the nodes remaining in \( G' \) at the end will be exactly those that belong to \( I \).

4. (a) If the problem is undecidable, then any attempt to solve it with an algorithm will lead to the algorithm producing incorrect results or looping on some inputs. Since this is usually not acceptable behaviour, the application in this form cannot be implemented. The problem needs to be reformulated, for example by restricting it to some special cases where it can be solved.

(b) If the problem is \( \text{NP} \)-complete, there is no point in looking for a polynomial time algorithm, since finding one is exactly as difficult as proving \( P = \text{NP} \). However, we know that exponential time algorithms do exist, and may well be good enough if we don’t need to consider very large inputs. It is also possible that we are lucky and the application only needs “easy” inputs where the algorithm runs fast.

For \( \text{NP} \)-complete problems (or more precisely the corresponding search problems, see problem 3) it is also common to use algorithms that only give an approximately optimal solution, or randomised algorithms that sometimes give better, sometimes worse solutions (e.g. simulated annealing).
An important difference between the cases (a) and (b) is that for NP-complete problems we know that at least small instances can be solved quickly. In contrast, for an undecidable problem even short and seemingly simple inputs may be problematic. For example, it is a famous open problem whether the following simple procedure halts on all inputs $n$:

```
COLLATZ(n)
    while $n > 1$
        do if $n$ is even
            then $n \leftarrow n/2$
        else $n \leftarrow 3n + 1$
```

(For additional information, search “Collatz conjecture”).