Basic problems:

Problem 2 should be solvable using the hint given in the original instructions. For Problem 4, reviewing the lecture slides for \textit{NP-completeness} will be helpful. For Problems 1 and 3, some hints to get you started are provided below.

1. Give a dynamic programming algorithm for the \textit{shortest reliable path} problem:

\textit{Input:} A graph \( G \) with lengths on the edges, two vertices \( s \) and \( t \) of \( G \), and a nonnegative integer \( k \).

\textit{Output:} A shortest path from \( s \) to \( t \) that uses at most \( k \) edges, or assert that no such path exists.

\textbf{Hint:} Some clarification for the above: Let \( P(s, t) \) be a path from \( s \) to \( t \), and \( e \in P(s, t) \) be the edges on the path. We’re looking for paths such that \(|e \in P(s, t)| \leq k \) (using at most \( k \) edges) to minimise \( \sum_{e \in P(s, t)} \ell(e) \) (the overall length of the path). For your dynamic-programming relation, you could consider the lengths of paths from the starting node \( s \) to any \( v \in V(G) \) comprising at most \( i \) edges.

Once you’ve decided on a dynamic-programming algorithm, consider a case where some of the edge lengths \( \ell(e) \) may be negative. Does your algorithm work in such a case?

3. [Dasgupta et al., Ex. 6.2] You are going on a long trip. You start on the road at mile post 0. Along the way there are \( n \) hotels, at mile posts \( a_1 < a_2 < \cdots < a_n \), where each \( a_i \) is measured from the starting point. The only places you are allowed to stop are at these hotels, but you can choose which of the hotels you stop at. You must stop at the final hotel (at distance \( a_n \)), which is your destination.

You’d ideally like to travel 200 miles a day, but this may not be possible (depending on the spacing of the hotels). If you travel \( x \) miles during a day, the \textit{penalty} for that day is \((200 - x)^2\). You want to plan your trip so as to minimize the total penalty—that is, the sum, over all travel days, of the daily penalties.

Give an efficient algorithm that determines the optimal sequence of hotels at which to stop.

\textbf{Hint:} For your dynamic-programming relation, consider the penalties for trips that start at mile post 0, and end at mile post \( a_k \).

Advanced problems:

5. [Dasgupta et al., Ex. 6.21] A \textit{vertex cover} of a graph \( G = (V, E) \) is a subset of vertices \( S \subseteq V \) that includes at least one endpoint of every edge in \( E \). Give a linear-time algorithm for the following task.

\textit{Input:} An undirected tree \( T = (V, E) \).

\textit{Output:} The size of the smallest vertex cover of \( T \).
For instance, in the following tree, possible vertex covers include \{A, B, C, D, E, F, G\} and \{A, C, D, F\} but not \{C, E, F\}. The smallest vertex cover has size 3: \{B, E, G\}.

**Solution:** This solution is similar to the 'Independent sets in trees' solution.

One dfs or bfs run in \(O(|V|)\) can arrange the tree as a rooted tree such that all nodes have a unique parent (except the root). Each node \(v \in V\) now defines a unique subtree \(T(v)\) hanging from it (consider removing the edge between the node and its parent). In the example tree, if \(B\) is the root then \(T(E)\) contains nodes \{E, D, F, G\}.

Let \(S(v)\) be the size of a smallest vertex cover in the subtree \(T(v)\), and refer to an example of such cover by \(\gamma(v)\). We find \(S(v)\) by dynamic programming all the way to \(S(v_{\text{root}})\).

For each \(v \in V\), a minimum vertex cover \(\gamma(v)\) for the subtree \(T(v)\) either contains \(v\) or it doesn’t contain \(v\). In the first case with \(v \in \gamma(v)\), one may consider erasing the edges adjacent to \(v\) as already taken care of, and one needs to look only at the forest of subtrees \(T(c)\) for \(c \in C(v)\). Now the vertex cover \(\gamma(v)\) must also contain (as subsets) some minimal vertex covers \(\gamma(c)\) for the children’s subtrees.

In the latter case with \(v \notin \gamma(v)\), the edges between \(v\) and each \(c \in C(v)\) force each child to be in the vertex cover \(\gamma(v)\). Erasing the now-covered edges, the only remaining part is the forest of grandchildren’s \(g_c \in GC(v)^c\) subtrees \(T(g_c)\). Then \(\gamma(v)\) also contains minimal vertex covers \(S(g_c)\) for each \(g_c \in GC(v)^c\).

Picking the better of these two cases leads to the recursion:

\[
S(v) = \min\{1 + \sum_{c \in C(v)} S(c), \quad |C(v)| + \sum_{g_c \in GC(v)} S(g_c)\}.
\]

Note that for leaves \(v_l\) this formula gives directly the correct value \(S(v_l) = 0\).

A recursive implementation is given as a pseudocode. Note that we save running time by keeping values of \(S(v)\) in a table and making a call only when the value is not calculated yet.
Algorithm 1: Algorithm for minimum vertex cover in trees

1 function min-vertexcover(v);
   Input: Undirected tree $T = (V, E)$
   Output: Size of minimum vertex cover in $T$
2 Select a root $v_0 \in V$;
3 Use dfs from $v_0$ to find a parent for each $v \in V \setminus \{v_0\}$;
4 $S[] \leftarrow$ empty table with index set $V$;
5 for each $v \in V$ do
6     $S[v] \leftarrow$ unknown;
7 end
8 evaluate-size($v_0$, $S$);
9 return $S[v_0]$;

10 ;
11 function evaluate-size($v$, $S$);
12 for $c \in C(v)$ do
13     if $S(c)$ unknown then
14         evaluate-size($c$, $S$);
15     end
16 end
17 value-self $\leftarrow 1 + \sum_{c \in C(v)} S[c]$;
18 value-noself $\leftarrow |C(v)| + \sum_{g \in GC(v)} S[g]$;
19 $S[v] \leftarrow \min\{\text{value-self}, \text{value-noself}\}$;
20 return ;

6. Prove the following basic facts about polynomial-time reducibility and complexity classes:

(a) If $S \leq^P T$ and $T \leq^P U$, then $S \leq^P U$.
(b) If $S \leq^P T$ and $T \in \text{P}$, then $S \in \text{P}$.
(c) Let $T$ be an $\text{NP}$-complete problem. If $T \in \text{P}$, then $\text{P} = \text{NP}$.
(d) Let $S$ be some $\text{NP}$-complete problem, $T \in \text{NP}$ and $S \leq^P T$. Then also $T$ is $\text{NP}$-complete.

Solution:

Recall the definition of $S \leq^P T$: If $S \leq^P T$ then there exists a polynomial-time computable function $f$ s.t. for any $s \in S$, $f(s) \in T$.

(a) For any $s \in S, f(s) \in T$ and $g(f(s)) \in U$ for some polynomial-time computable $f, g$. Thus $S \leq^P U$, using $g(f)$ as the polynomial-time computable function.
(b) Any $s \in S$ can be solved in polynomial time by solving $f(s)$ in polynomial time, where $f$ is a polynomial-time computable function. Thus, any $s \in S$ can be solved in polynomial time and $S \in \text{P}$.
(c) Recall the definition of $\text{NP}$-completeness: A problem class $T$ is $\text{NP}$-complete if $T \in \text{NP}$ and $S \leq^P T$ for all $S \in \text{NP}$.

Now, we know that $\text{P} \subset \text{NP}$ as any problem in $\text{P}$ will have a polynomial-time verification algorithm. It suffices to prove that $\text{NP} \subset \text{P}$. Now, for any problem $S \in \text{NP}$ and $s \in S$, we have $f(s) \in T$ for some polynomially-computable function $f$. As $T \in \text{P}$, $S \in \text{P}$ and $\text{NP} \subset \text{P}$.
(d) We know that $T \in \text{NP}$, so it suffices to show that $R \leq^p T$ for all $R \in \text{NP}$. As $S$ is $\text{NP}$-complete, we know that for all $r \in R$, $f(r) \in S$ for some polynomial-time computable $f$. As $S \leq^p T$, $g(f(r)) \in T$ for some polynomial-time computable $g$. Hence, $R \leq^p T$ for all $R \in \text{NP}$. 