1 Beware of algorithms carrying oracles.Consider the following optimization problems, and for each one of them do the following:

(I) (2 PTS.) State the natural decision problem corresponding to this optimization problem.

(II) (3 PTS.) Either: (A) prove that this decision problem is NP-COMPLETE by showing a reduction from one of the NP-COMPLETE problems seen in class (if you already seen this problem in class state "seen in class" and move on with your life). (B) Alternatively, provide an efficient algorithm to solve this problem.

(III) (5 PTS.) Assume that you are given an algorithm that can solve the **decision** problem in polynomial time. Show how to solve the original optimization problem using this algorithm in polynomial time, and output the solution that realizes this optimal solution.

(A) (10 PTS.)

NO COVER

Instance: Collection $\mathcal{C}$ of subsets of a finite set S.
Target: Compute the maximum k, and the sets $S_1, \ldots, S_k$ in $\mathcal{C}$, such that $S \not\subseteq \cup_{i=1}^{k} S_i$.

Solution:

(I) (2 PTS.) **Decision problem:** Given k in addition, decide if there is a valid solution of size k.

(II) (3 PTS.) **NPC or polynomial?** This problem is polynomial. If there such a partial cover, then there must be at least one element $x \in S$ that is not covered. For such an x, take all the sets in $\mathcal{C}$ that do not cover X. Putting it differently, scan all the sets of $\mathcal{C}$, and count for every element of S how many times it is being covered. Take the element y that is covered the smallest number of times, and let $\mathcal{U} \subseteq \mathcal{C}$ be all the sets that do not contain y, and return $k := |\mathcal{U}|$.
The running time is proportional to the size of the input. (The input being a list of sets, each set being a list of elements in the set.)

(III) (5 PTS.) **Reduction of optimization to decider.**
Since the problem is solvable in linear time, this is not interesting in this case.

(B) (10 PTS.)

TRIPLE HITTING SET

Instance: A ***ground set*** $U = \{1, \ldots, n\}$, and a set $\mathcal{F} = \{U_1, \ldots, U_m\}$ of subsets of U.
Target: Find the smallest set $S' \subseteq U$, such that S' hits all the sets of $\mathcal{F}$ at least three times. Specifically, $S' \subseteq U$ is a ***triple hitting set*** if for all $U_i \in \mathcal{F}$, we have that S' contains at least three elements of U_i.

Solution:

(I) (2 PTS.) **Decision problem:**

TRIPLE HITTING SET

Instance: A ***ground set*** $U = \{1, \ldots, n\}$, and a set $\mathcal{F} = \{U_1, \ldots, U_m\}$ of subsets of U. And a parameter k.
Question: Is three a set $S' \subseteq U$ of size k that triple hit $\mathcal{F}$?

(II) (3 PTS.) **NPC or polynomial?**

The problem is NPC. First, it is easy to verify that is in NP (given a solution, just verify it intersects the given solution sufficient number of times.).
As for the reduction, it is from

HITTING SET

Instance: A ***ground set*** $U = \{1, \ldots, n\}$, and a set $\mathcal{F} = \{U_1, \ldots, U_m\}$ of subsets of U. And a parameter k.
Question: Is three a set $S' \subseteq U$ of size k that intersects all the sets of $\mathcal{F}$?

This problem is NPC because it is straightforward restatement of SET COVER (verify!). Given an instance of HITTING SET, add $n+1$ and $n+2$ to U, and add them also to all the sets of $\mathcal{F}$. Clearly, a hitting set of size k of the original set corresponds to a triple hitting set of size $k+2$ of the new instance, and this holds also in other direction – if there is a triple hitting set of size $k+2$ of the resulting instance of TRIPLE HITTING SET, then one can assume that $n+1$ and $n+2$ are in the solution, and one can remove them, and get a valid hitting set for the original instance of size k.

(III) (5 PTS.) **Reduction of optimization to decider.**
First, using the decider, using binary search, find the minimum value k of the optimal solution. Now, for every $i \in U$, try to remove it from U (and remove it from all the sets that contain it, naturally), and if it the resulting instance still has a valid solution (using the decider). If not, restore the instance, otherwise, continue to the next number. Clearly, in the end of this process, the only numbers remaining in U are the desired solution. This algorithm requires $n + O(\log n)$ calls to the decider, and it is clearly polynomial time otherwise.

(C) (15 PTS.)

Max Inner Spanning Tree

Instance: Graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$.
Target: Compute the spanning tree T in G where the number of vertices in T of degree two or more is maximized.

Solution:

(I) (2 PTS.) **Decision problem:**

Inner Spanning Tree

Instance: Graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$ and k.
Question: Compute the spanning tree T in G where the number of vertices in T of degree two or more is at least k.

(II) (3 PTS.) **NPC or polynomial?**
This problem is NPC via a reduction from Hamiltonian Path.
The problem is clearly in NP- given a candidate spanning tree, it is straightforward to count the number of its internal nodes, and compare it to k.
Now, the only tree that has two leafs, is a path, and such a tree has $n-2$ interval vertices of degree 2.
As such, given an instance G of Hamiltonian Path with n vertices, this reduces to an instance of Inner Spanning Tree by adding the parameter $k = n-2$ to the given instance.

(III) (5 PTS.) **Reduction of optimization to decider.**
Using binary search and the decider, find the value of k in the optimal solution. Now, as above, repeatedly try to remove every edge in the graph, and check if the

remaining graph still have an optimal solution of the desired value by calling the decider. By the end of this process, what remains is the desired optimal solution.

(D) (15 PTS.)

Cover by paths (edge disjoint).

Instance: Graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$.
Target: Compute the minimum number k of paths $\pi_1, \ldots, \pi_k$ that are edge disjoint, and their union cover all the edges in G.

Solution:

(I) (2 PTS.) **Decision problem:**

Cover by paths (edge disjoint).

Instance: Graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$ and a number k.
Target: Are there k paths $\pi_1, \ldots, \pi_k$ that are edge disjoint, and their union cover all the edges in G.

(II) (3 PTS.) **NPC or polynomial?**
This problem is polynomial. Indeed, the compute the number t of odd degree vertices in the graph G. This number must be even (verify you know how to prove this fact!), and return $t/2$ as the desired number.

Lemma 0.1. *The above algorithm is correct.*

Proof: The proof is constructive by induction. If $t = 2$, then the graph is Eulerian, and one can compute an Eulerian path in G that visits all the vertices of G in linear time (this is well known, and I assume you already know this).
Otherwise, let u, v be the two odd degree vertices that are in the same connected component of G, and let π be their shortest path. Consider the graph G' resulting from removing from it all the edges of π. The G' has $t - 2$ odd degree vertices, and such, by induction, its edges can be covered by $(t-2)/2$ edge disjoint paths. Adding π to this cover, results in the desired cover. ■

(III) (5 PTS.) **Reduction of optimization to decider.**
Not interesting in this case.

2 Reduction, deduction, induction, and abduction.

The following question is long, but not very hard, and is intended to make sure you understand the following problems, and the basic concepts needed for proving NP-Completeness.

All graphs in the following have n vertices and m edges.

For each of the following problems, you are given an instance of the problem of size n. Imagine that the answer to this given instance is "yes", and that you need to convince somebody that indeed the answer to the given instance is `yes`. To this end, describe:

(I) An algorithm for solving the given instance (not necessarily efficient). What is the running time of your algorithm?
(II) The format of the proof that the instance is correct.
(III) A bound on the length of the proof (its have to be of polynomial length in the input size).
(IV) An efficient algorithm (as fast as possible [it has to be polynomial time]) for verifying, given the instance and the proof, that indeed the given instance is indeed `yes`. What is the running time of your algorithm?

2.A. (5 PTS.)

Semi-Independent Set

Instance: A graph G, integer k
Question: Is there a semi-independent set in G of size k? A set $X \subseteq \mathsf{V}(\mathsf{G})$ is semi-independent if no two vertices of X are connected by an edge, or a path of length 2.

Solution:

(I) **Solver**: Compute for every pair of vertices in the graph if they are in distance 1 or 2 (in edge distance). This can be done by doing BFS for each vertex, thus taking $O(nm)$ time. Build a graph $\mathsf{H} = (\mathsf{V}(\mathsf{G}), \mathsf{E}')$ where a pair of vertices is connected if and only if there in distance 1 or 2 from each other in G. Clearly, the task in hand is to find an independent set in H of size k.
So, enumerate all subsets of vertices of $\mathsf{V} = \mathsf{V}(\mathsf{G})$ of size k. This takes $\binom{n}{k}$ time with careful implementation (verify you know how to do this!). Now, check if it is independent in G – that takes $O(n^2)$ time. Let k' be the largest independent set in H found by this process. If $k' \geq k$ return YES, otherwise return NO. The running time of the algorithm is $O\left(\binom{n}{k} n^2 + nm\right) = O(n^k + n^3)$.

(II) **Proof format:** A list of k integer numbers between 1 and n (i.e., assume $\mathsf{V} = \{1, \ldots, n\}$).

(III) **Proof length:** $O(k) = O(n)$ words. In bits, the length of the encoding is $O(k \log n)$.

(IV) **Certifier:** Given a list L of k vertices, add an extra vertex u to G, and connect it to all the vertices of L. Do a **BFS** from u, and let $d(v)$ denote the distance from u in the BFS tree. Now, scan all the edges of the graph. An edge xy, that is not in the BFS tree edge, that connects two vertices of L, or connects a vertex of L to a vertex of depth 2 in the BFS tree both testify that the independence condition fails. As such, this can be checked in $O(m)$ time.

2.B. (5 PTS.)

3EdgeColorable

Instance: A graph G.
Question: Is there a coloring of the edges of G using three colors, such that no two edges of the same color are adjacent?

Solution:

(I) **Solver**: If the graph has a vertex with more than three edges, the answer is NO, since the is no valid coloring for the edges around this vertex. Thus, this graph has at most $3n$ edges (in fact, $(3/2)n$ as this count every edge twice).
Next, enumerate all possible 3 colorings of the edges of the graph. There are $3^{(3/2)n}$ such colorings, as $m \leq (3/2)n$. Now, for every vertex, count the number of edges adjacent to it of each color. If the count is larger than one for any of the colors, reject the coloring and move to the next candidate coloring. Clearly, this check can be done in $O(m+n) = O(n)$ time, and overall $O(n3^{(3/2)n})$ time.

(II) **Proof format:** A list of m numbers, each number is either $1, 2$ or 3.

(III) **Proof length:** $O(m) = O(n)$.

(IV) **Certifier:** As described in the solver part. Its running time is $O(n)$.

2.C. (5 PTS.)

Subset Sum

Instance: S: Set of positive integers. t: An integer number (target).
Question: Is there a subset $X \subseteq S$ such that $\sum_{x \in X} x = t$?

Solution:

Assume $S = \{s_1, \ldots, s_n\}$.

(I) **Solver**: Enumerate all possible subsets of S, and check for each such subset if it sum is t. This takes $O(2^n n)$ time.

(II) **Proof format:** The certificate is a binary string $b_1, \ldots, b_n$, where b_i is one if and only if the ith number is supposed to be in the solution subset.

(III) **Proof length:** So the certificate length is $O(n)$.

(IV) **Certifier:** The certifier adds up the numbers with $b_i = 1$, and check that the sum is the desired quantity. This takes $O(n)$ time.

2.D. (5 PTS.)

3DM

Instance: X, Y, Z sets of n elements, and T a set of triples, such that $(a, b, c) \in T \subseteq X \times Y \times Z$.
Question: Is there a subset $S \subseteq T$ of n disjoint triples, s.t. every element of $X \cup Y \cup Z$ is covered exactly once.?

Solution:

Assume $X = \{x_1, \ldots, x_n\}$, $Y = y_1, \ldots, y_n$ and $Z = \{z_1, \ldots, z_n\}$. Furthermore, the triplets are $T = \{t_1, \ldots, t_n\}$, where t_i is of the form $(i_1, i_2, i_3) \in \{1, \ldots, n\}^3$.

(I) **Solver**: Let $m = |T|$. Enumerate all possible subsets of T of size n, there are $\binom{m}{n} = O(m^n)$ such subsets. For each subset verify that all the triplets in it are disjoint, and if so they form a valid solution. If so, return this as a valid solution.
Otherwise, the algorithm returns false.
the running time of the algorithm is $O(m^n n)$ time.

(II) **Proof format:** The certificate is as such a list of n numbers $i_1, \ldots i_n$ in the range $\{1, \ldots, m\}$.

(III) **Certificate length:** The length of certificate is $O(n)$.

(IV) **Certifier:** The certifier checks that all the indices are distinct (by sorting the numbers in $O(n)$ time [using radix sort]). Next, it checks that the first coordinate of all the triplets $t_{i_1}, \ldots, t_{i_n}$ are distinct (by again, say, using radix sort). It repeat this two the other two coordinates. If everything goes through then this is a valid solution.
As such, the verification takes $O(n)$ time.

2.E. (10 PTS.)

SET COVER

Instance: $(\mathsf{U}, \mathcal{F}, k)$:
U: A set of n elements
$\mathcal{F}$: A family of m subsets of S, s.t. $\bigcup_{X \in \mathcal{F}} X = \mathsf{U}$.
k: A positive integer.
Question: Are there k sets $S_1, \ldots, S_k \in \mathcal{F}$ that cover S. Formally, $\bigcup_i S_i = \mathsf{U}$?

Solution:

Assume $\mathsf{U} = \{1, \ldots, n\}$, and $\mathcal{F} = \{F_1, \ldots, F_m\}$, where $F_i \subseteq \mathsf{U}$ is defined by a binary vector of length n, where the jth bit decides if $j \in F_i$.

(I) **Solver**: Enumerate all possible subsets of the sets of size k. There are $O(m^k)$ such sets. Now, for each such set of k sets of $\mathcal{F}$, verify that they cover the ground set. This takes $O(kn)$ time, as described below. As such, overall, the running time is $O(m^k kn)$.
(II) **Proof format:** The certificate is a list of k numbers $n_1, \ldots, n_k \in \{1, \ldots, m\}$.
(III) **Proof length:** $O(k)$.
(IV) **Certifier:** The certifier computes the bitwise-or of the vectors $F_{n_1}, \ldots, F_{n_k}$ and verifies that the resulting vector has 1 in all coordinates. This takes $O(kn)$ time.

2.F. (10 PTS.)

CYCLE HATER.

Instance: An undirected graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$, and an integer $k > 0$.
Question: Is there a subset $X \subseteq \mathsf{V}$ of at least k vertices, such that no cycle in G contains any vertex of X.

Solution:

(I) **Solver**: One can check for a vertex v of the graph if it is in a cycle in linear time in the size of the graph. This can example be done by doing a DFS from v, and seeing if encounter any back edge going back into v. Overall, doing this for all vertices, this takes $O(nm)$ time (this can be done in $O(n + m)$ by a more clever algorithm, by discovering all bridges in the graph).
Let U be the set of all vertices that are not marked as belonging to a cycle. If $|U| \geq k$ then return YES, otherwise, return NO.
(II) **Proof format:** A list of k vertices.
(III) **Proof length:** $O(k)$
(IV) **Certifier:** Check, using DFS as described above, for each of these k vertices that they are not in a cycle. This takes $O(km)$ time.

2.G. (10 PTS.)

Many Meta-Spiders.

Instance: An undirected graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$ and an integer k.
Question: Are there k vertex-disjoint meta-spiders that visits all the vertices of G?

A ***meta-spider*** in a graph G is defined by two vertices u, v (i.e., the head and tail of the meta-spider), and a collection Π of simple paths all starting in v and ending at u, that are vertex disjoint (except for u and v). The vertex set of such a spider is all the vertices that the paths of Π visit (including, of course, u and v).

Solution:

(I) **Solver**: The solver here would generate all candidate solutions, and would check if any of them is valid, using the certifier.
Every path in a meta-spider must visit at least one vertex except for u and v. As such, in a valid solution, there are at most $T \leq n - 1$ paths in all the spiders. Every such path, can be described as sequence starting at a head vertex, and ending at a tail vertex. As such, the total length of these paths is at most $2T + n$. As such, there are at most $2^{2T+n} n^{2T+n}$ such possible collection of paths, and this is the time it takes to generate them. Specifically, consider a sequence of numbers $x_1, b_1, x_2, b_2, \ldots, x_u, b_u$, where the x_is are in V, and $b_i \in \{0, 1\}$. We use the convention, that $x_i = 1$, indicates that a new path start at this location.
There are as such $O(2^{3n} n^{3n})$ certificates being tried, and each one of them can be tested in $O(n)$ time. Thus, yielding the desired running time.

(II) **Certificate format:** The format is described above: It is a sequence of at most $3n$ pairs x_i, b_i, where x_i is a vertex number, and b_i is a bit.

(III) **Certificate length:** $O(n)$ words. $O(n \log n)$ bits.

(IV) **Certifier:** Given a collection of paths as encoded above, the certifiers works as follows:

1. Verify that every vertex is covered by at least one of these paths. Time: $O(n)$.
2. Every encoded path uses edges that exists in the graph. This can be checked in $O(1)$ time, after converting the graph to adjacency graph representation.
3. Verify that there are at most $2k$ vertices that are covered more than once by these paths, and these are only start/ending vertices of these path. Also verify that paths are faithful and do not connect heads/tails belonging to other meta-spiders.. Time: $O(n)$.

Thus, the certificate runs in $O(n)$ time.