1. The traveling salesman problem can be defined in two ways:

   - The Traveling Salesman Problem
     - INPUT: A weighted graph G
     - OUTPUT: Which tour $(v_1, v_2, \ldots, v_n)$ minimizes $\sum_{i=1}^{n-1}(d[v_i, v_i+1]) + d[v_n, v_1]$
   - The Traveling Salesman *Decision* Problem
     - INPUT: A weighted graph G and an integer k
     - OUTPUT: Does there exist and TSP tour with cost $\leq k$

   Suppose we are given an algorithm that can solve the traveling salesman decision problem in (say) linear time. Give an efficient algorithm to find the actual TSP tour by making a polynomial number of calls to this subroutine.

Solution: We will first find the minimal cost of a TSP tour, and then find the minimal TSP tour using the minimal cost. We let $TSPD(G, k)$ denote the result of running the given algorithm for traveling salesman decision problem on the input (G, k).

Finding the minimal cost : Let d_{min} and d_{max} denote the minimal and maximal weight of the edges in E, respectively. Since a TSP tour would contain $|V|$ edges, we can infer that the cost of any TSP tour would be lower-bounded by $|V|d_{min}$ and upper-bounded by $|V|d_{max}$. Therefore, we can run binary search on the integers ranging from $|V|d_{min}$ to $|V|d_{max}$ to find the integer value k such that $TSPD(G, k)$ is True and $TSPD(G, k-1)$ is False. The integer k would be the minimal cost of a TSP tour.

Finding the minimal TSP tour : Let k be the minimal cost of a TSP tour in G. Let e be an arbitrary edge of G and let G' be a graph obtained by removing e from G. Suppose e was not included in a minimal TSP tour T of G(note that minimal TSP tour is not necessarily unique). Then $TSPD(G', k)$ would return True, since the minimal TSP tour T still remains in G'. Now suppose e was included in every minimal TSP tour of G. Then $TSPD(G', k)$ would return False, since without e we cannot formulate a minimal TSP tour with the cost k. Therefore, for each $e \in E$, we will remove e from G and run $TSPD(G, k)$. If $TSPD(G, k)$ returns True, that means there exists a minimal TSP in G that does not contain e, so we will leave e removed. If $TSPD(G, k)$ returns False, that means e is definitely a part of any minimal TSP in G, so we will add e back into G. Once we have all unnecessary edges removed, we can run a linear time search algorithm from any node to construct the tour. The following is the pseudocode of the algorithm.

```
TSP(G(V,E)):
    lower ← |V| * d_min
    upper ← |V| * d_max
    k ← BinSearch(lower, upper)
    for e in E:
        remove e from G
        if TSPD(G,k)=False:
            add e to G
    s ← ArbitraryNode(G)
    tour ← DFS(s)

    return tour
```

```
BinSearch(lower, upper):
    if upper − lower < 1000
        find k via brute force
    else
        mid ← (lower+upper)/2
        if TSPD(G, mid) is true
            k ← BinSearch(lower,mid)
        else
            k ← BinSearch(mid+1,upper)
    return k
```

The complexity of the binary search is dependent on the difference between the upper and lower bound of the total cost. That is, if we let $c = d_{max} - d_{min}$, the complexity of the binary search is $O(\log(c|V|))$. However, note that $c = O(2^n)$ where n is the length(number of bits) of the input, since with n bits you can only represent integers upto 2^n. Therefore, the complexity of the binary search is $O(\log(c|V|)) = O(\log c) + O(\log|V|) = O(n)$, which is linear in the input size. The rest of the algorithm loops over every edge, and TSPD is assumed to be linear ($O(V+E)$), so the runtime of the tour construction is $O(E(E+V))$. Therefore, the total runtime would be $O(n+E(E+V))$ which is polynomial in input size. For grading, the reasoning for the complexity of binary search is not required; providing $O(E(E+V))$ and reasoning about the tour construction is sufficient for full marks.

■

2. For any integer k, the problem kSAT is defined as follows:

- INPUT: A boolean formula Φ in conjunctive normal form, with exactly k distinct literals in each clause.
- OUTPUT: TRUE if Φ has a satisfying assignment, and FALSE otherwise.

(a) Describe and analyze a polynomial-time reduction from 2SAT to 3SAT, and prove your reduction is correct.

Solution: One such reduction (of infinitely many possible ones) is as follows. Let

$$\Phi = \bigwedge_{i=1}^{n} \ell_{i,1} \vee \ell_{i,2}$$

be the instance to 2SAT; in the above description of Φ, $\ell_{i,1}$ and $\ell_{i,2}$ are *literals* for all $1 \le i \le n$, not variables. Construct 3CNF formula

$$\Phi' = \bigwedge_{i=1}^{n} (\ell_{i,1} \vee \ell_{i,2} \vee x) \wedge (\ell_{i,1} \vee \ell_{i,2} \vee \overline{x});$$

here, x is a variable *not* in Φ. Input Φ' into the black box algorithm $\mathscr{A}$ for 3SAT, and feed the output of $\mathscr{A}$ as the output of the constructed algorithm for 2SAT. Φ' has exactly twice the number of clauses as Φ and there are at most $2n$ variables. Thus, Φ' can be constructed by brute force in ***time $O(n)$*** by a scanning through once Φ. The reduction is linear-time and thus polynomial-time.

We now prove the correctness of this reduction by proving the following claim: Φ has a satisfying assignment $\iff$ Φ' has a satisfying assignment.

- $\Rightarrow$ Suppose there is an assignment A of the variables in Φ that makes Φ evaluate to TRUE. Fix $1 \le i \le n$. By the definition of $\wedge$, we have that $\ell_{i,1} \vee \ell_{i,2}$ evaluates to TRUE under A. By the definition of $\vee$, this gives either $\ell_{i,1} = \text{TRUE}$ or $\ell_{i,2} = \text{TRUE}$ under A. Define the assignment A' as one that coincides with A for variables in Φ and assigns *any* truth value to x. By the definition of $\vee$, both $\ell_{i,1} \vee \ell_{i,2} \vee x$ and $\ell_{i,1} \vee \ell_{i,2} \vee \overline{x}$ evaluate to TRUE under A'. Since this analysis holds for all $1 \le i \le n$, by the definition of $\wedge$, we have that $\bigwedge_{i=1}^{n} (\ell_{i,1} \vee \ell_{i,2} \vee x) \wedge (\ell_{i,1} \vee \ell_{i,2} \vee \overline{x})$ evaluates to TRUE under A'. But $\bigwedge_{i=1}^{n} (\ell_{i,1} \vee \ell_{i,2} \vee x) \wedge (\ell_{i,1} \vee \ell_{i,2} \vee \overline{x}) = \Phi'$, which implies Φ' has a satisfying assignment.
- $\Leftarrow$ Suppose there is an assignment A' of the variables in Φ' that makes Φ' evaluate to TRUE. Fix $1 \le i \le n$. By the definition of $\wedge$, $(\ell_{i,1} \vee \ell_{i,2} \vee x) \wedge (\ell_{i,1} \vee \ell_{i,2} \vee \overline{x})$ evaluates to TRUE under A'. By the definition of $\wedge$ again, $\ell_{i,1} \vee \ell_{i,2} \vee x$ and $\ell_{i,1} \vee \ell_{i,2} \vee \overline{x}$ both evaluate to TRUE under A'. It can easily be seen that both x and $\overline{x}$ cannot be TRUE under A'. Assume that x is TRUE under A' without loss of generality. Then $\overline{x}$ evaluates to FALSE, which implies that either $\ell_{i,1}$ or $\ell_{i,2}$ must evaluate to TRUE. We prove this by contradiction. Suppose both $\ell_{i,1}$ and $\ell_{i,2}$ evaluate to FALSE. Then $\ell_{i,1} \vee \ell_{i,2} \vee \overline{x}$ evaluates to FALSE, a contradiction. By the definition of $\vee$, then $\ell_{i,1} \vee \ell_{i,2}$ evaluates to TRUE under A' (and the restriction of the assignment A of A' to variables in Φ). Since this analysis holds for all $1 \le i \le n$, by the definition of $\wedge$, we have that

$\bigwedge_{i=1}^{n} \ell_{i,1} \vee \ell_{i,2}$ evaluates to TRUE under A. But $\bigwedge_{i=1}^{n} \ell_{i,1} \vee \ell_{i,2} = \Phi$, which implies Φ has a satisfying assignment.

■

(b) Describe and analyze a polynomial-time algorithm for 2SAT. *[Hint: This problem is strongly connected to topics earlier in the semester.]*

Solution: Let

$$\Phi = \bigwedge_{i=1}^{n} \ell_{i,1} \vee \ell_{i,2}$$

be the instance to 2SAT; in the above description of Φ, $\ell_{i,1}$ and $\ell_{i,2}$ are *literals* for all $1 \leq i \leq n$, not variables. Construct a *directed* graph $G = (V, E)$ as follows:

- x is a variable in $\Phi \iff x, \overline{x} \in V$
- $\ell_1 \vee \ell_2$ is a clause for some *literals* ℓ_1 and ℓ_2 in $\Phi \iff \overline{\ell}_1 \to \ell_2, \overline{\ell}_2 \to \ell_1 \in E$

Compute the strong components of G using Kosaraju's algorithm and check if, for any variable x, x and $\overline{x}$ are in the same strong component. If so, return FALSE. Otherwise, return TRUE. Kosaraju's algorithm and checking the above condition combined require time $O(V + E)$ *in terms of the graph* G. Since $V \leq 2n$ and $E \leq 2n$ where n is the number of clauses in Φ, in terms of the original input Φ, this algorithm requires ***time $O(n)$***. This verifies that the algorithm is indeed polynomial-time. ■

(c) Why don't these results imply a polynomial-time algorithm for 3SAT?

Solution: We do not have enough information. It's worth noting that either of the following changes to the prompts of parts (a) and (b) would imply a polynomial-time algorithm for 3SAT:

- Part (a) asks for polynomial-time reduction from 3SAT to 2SAT instead of from 2SAT to 3SAT.
- Part (b) asks for a polynomial-time algorithm for 3SAT instead of 2SAT.

Also, just because you can use a harder problem (in this case 3SAT) to solve an easier one (in this case 2SAT) doesn't mean that is the *only* way to solve 2SAT (as you can see in part (b)). This is a subtle but very important distinction that is at the core of reductions. ■

3. A disjunctive normal form (DNF) formula is the converse of CNF; i.e., it is an $\vee$ of a number of clauses where each clause is an $\wedge$ of some terms. E.g: $(x \wedge y \wedge z) \vee (z \wedge y \wedge \neg w) \vee (x \wedge \neg z)$. DNF-SAT is the analog problem of (CNF-)SAT: given a DNF formula f, determine if there is a satisfying assignment of the corresponding variables that renders the formula true.

   (a) Design and analyze an efficient algorithm for DNF-SAT. *Hint: DNF-SAT can be solved directly, a reduction is not needed.*

   Solution: A DNF formula f is an $\vee$ of a number of clauses. How to determine if f can be evaluated to true? If one of the clauses can be evaluated to true, then f can be evaluated to true (f is satisfiable).

   For DNF, a clause is an $\wedge$ of some terms. How to determine if a clause can be evaluated to true? A clause cannot be evaluated to true if it contains both a term x and its complement $\neg x$.

   The algorithm goes as follows:

   - Initalize list of variable with blank spaces that will denote their truth assignment.
   - for each clause in the DNF formula:
     - mark the variable according to the literals in the clause.
     - if a variable is already marked in a way that does not correspond to the literal, then return false
   - return true

   This algorithm takes $O(|f|)$ time to run because it checks the membership of $\neg$term for every term in f. ■

   (b) Demonstrate a reduction from 3SAT to DNF-SAT and analyze its runtime. (Hint: use the distributive law.) (Another hint: this reduction will not be efficient, it will not be a polynomial-time reduction).

   Solution: Both 3SAT and DNF-SAT want to check satisfiability. If we can convert an arbitrary 3CNF formula f into an equivalent DNF formula, then we can use the algorithm for DNF-SAT to solve 3SAT.

   An 3CNF formula f is an $\wedge$ of a number of clauses where each clause is an $\vee$ of three terms. Consider an example with two clauses: $(x \vee y \vee z) \wedge (\neg x \vee y \vee \neg z)$. Using the distributive law, we can convert the 3CNF into an equivalent DNF with nine clauses where each clause has two terms in it: $(x \wedge \neg x) \vee (x \wedge y) \vee (x \wedge \neg z) \vee (y \wedge \neg x) \vee (y \wedge y) \vee (y \wedge \neg z) \vee (z \wedge \neg x) \vee (z \wedge y) \vee (z \wedge \neg z)$.

   In general, for a 3CNF formula f with n clauses, we can use the distributive law to find an equivalent DNF with 3^n clauses where each clause has n terms in it. Therefore, the reduction from 3SAT to DNF-SAT takes $O(n3^n)$ time to run. ■

4. A **Hamiltonian cycle** in a graph is a cycle that visits every vertex exactly once. A **Hamiltonian path** in a graph is a path that visits every vertex exactly once, but it need not be a cycle (the last vertex in the path may not be adjacent to the first vertex in the path.)

   Consider the following three problems:

   - *Directed Hamiltonian Cycle* problem: checks whether a Hamiltonian cycle exists in a *directed* graph,
   - *Undirected Hamiltonian Cycle* problem: checks whether a Hamiltonian cycle exists in an *undirected* graph.
   - *Undirected Hamiltonian Path* problem: checks whether a Hamiltonian path exists in an *undirected* graph.

   (a) Give a polynomial time reduction from the *directed* Hamiltonian cycle problem to the *undirected* Hamiltonian cycle problem.

Solution: For any arbitrary directed graph $G_d := \{V_d, E_d\}$, construct the following undirected graph $G_u := \{V_u, E_u\}$:

- $V_u := \{v_{in}, v_{mid}, v_{out} \mid v \in V_d\}$. For each of the vertices in the directed graph, we split them into a triplet of in, mid, and out.
- $E_u := \{(u_{out}, v_{in}) \mid (u, v) \in E_d\} \cup \{(v_{in}, v_{mid}), (v_{mid}, v_{out}) \mid v \in V_d\}$. For each of the triplets that comes from the same vertex, we connect them in the order of in-mid-out. The directed edges in the V_d become the undirected ones that connect out and in between corresponding triplets.

Notice that $|V_u| = 3\,|V_d|$ and $|E_u| = |E_d| + 2\,|V_d|$, so this reduction is linear.

$\Rightarrow$: Suppose that in G_d there exists a Hamiltonian cycle $C_d := \left(c_1, c_2, \ldots, c_{|V_d|}\right)$, where $c_i \in V_d$. Then in G_u there should also exist

$$C_u := \left(c_{1in}, c_{1mid}, c_{1out}, c_{2in}, c_{2mid}, c_{2out}, \ldots, c_{|V_d|in}, c_{|V_d|mid}, c_{|V_d|out}\right),$$

which is a Hamiltonian cycle in G_u.

$\Leftarrow$: Suppose that in G_u there exists a Hamiltonian cycle C'_u. By definition, within each of the triplets there should only be a path of order in-mid-out, and between two triplets there should only be an edge of out-in. Thus, C'_u should always be of the following form

$$C'_u := \left(c'_{1in}, c'_{1mid}, c'_{1out}, c'_{2in}, c'_{2mid}, c'_{2out}, \ldots, c'_{|V_d|in}, c'_{|V_d|mid}, c'_{|V_d|out}\right),$$

which corresponds to a Hamiltonian cycle $C'_d := \left(c'_1, c'_2, \ldots, c'_{|V_d|}\right)$ in G_d.

■

(b) Give a polynomial time reduction from the *undirected* Hamiltonian Cycle to *directed* Hamiltonian cycle.

Solution: This reduction is simpler than the previous one. Given an instance G of undirected Hamiltonian cycle, Let G' be the directed graph with the same vertices as G and containing edges $u \to v$ and $v \to u$ for every edge $uv \in G$.

($\Rightarrow$) If C is a cycle in G, then C is also a cycle in the directed graph G'. For every $u \to v \in G'$, the edge uv is in G.

($\Leftarrow$) If C is a cycle in G', then C is also a cycle in the original graph G. For every $uv \in G$, the edge $u \to v \in G'$ by the construction.

■

(c) Give a polynomial-time reduction from undirected Hamiltonian *Path* to undirected Hamiltonian *Cycle*.

Solution: Let the input to this problem be an undirected graph G. The goal is to produce G' such that G has a Hamiltonian path if G' has a Hamiltonian cycle.

This can be done by adding a vertex v with edges to every vertex in the original graph G, this will be G'.

$\Rightarrow$: If there exists a Hamiltonian path P in G, starting with vertex s and ending with vertex t. Then $[v, s, P, t, v]$ is a Hamiltonian cycle in G'.

$\Leftarrow$: In the other case, if C is the Hamiltonian cycle in G', then removing v from C will return a Hamiltonian path in G. ■