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: There are 2 parts to this problem, finding the TSP minimum tour value, then finding the TSP minimum tour.

To find the minimum tour value we use binary search to find the integer value k where the decision problem returns true but $k-1$ returns false.

To find the minimum tour we pick a vertex then remove an edge then recheck the TSP decision problem for k. If the decision returns false then that edge has to be part of the minimum tour. If it returns true then that means the edge was not part of the minimum tour, keep removing edges until you get false. When this edge is found move on to the vertex it connects to. Repeat this process removing previous visited vertices from the neighbor sets.

Note: We keep removing edges in case there are multiple tours with the minimum value. Also we add the last vertex in because once we add the second to last there is only one option and there will be no more unvisited vertices when we check the neighbors of the last vertex.

```
TSP(G(V,E)):
  lower ← |V|*min(E)
  upper ← |V|*max(E)
  k ← BinSearch(lower,upper)
  v ← v_1
  visit ← empty
  while visit ≠ V
      check ← neighbors(v)
      for w in check-visit
          G ← G-e_vw
          if TSPD(G,k) is false and v is not in visit
              G ← G+e_vw
              add v to visit
              v ← w
      if |visit| = |V|-1
          add V-visit to visit
  return visit
```

```
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,upper)
  return k
```

The binary search is dependent on edge weights which if they are not too different will be inconsequential. This algorithm loops through the edges and TSPD is linear ($O(V+E)$), so the total running time is $O(E(V+E))$

■

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 **2**SAT to **3**SAT, 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 **2**SAT; 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 **3**CNF 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 **3**SAT, and feed the output of $\mathscr{A}$ as the output of the constructed algorithm for **2**SAT. 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 $1 \leq i \leq n$. 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 $\ell_{i,1} =$ TRUE or $\ell_{i,2} =$ 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 $1 \leq i \leq n$, 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 $1 \leq i \leq n$. 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 $1 \leq i \leq n$, 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 **2**SAT. *[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 **2**SAT; 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 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 $V \le 2n$ and $E \le 2n$ where n is the number of clauses in Φ, in terms of the original input Φ, this algorithm requires ***time*** $\boldsymbol{O(n)}$. This verifies that the algorithm is indeed polynomial-time. ■

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

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 **3**SAT:

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

Also, just because you can use a harder problem (in this case **3**SAT) to solve an easier one (in this case **2**SAT) doesn't mean that is the *only* way to solve **2**SAT (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 $G_u := \{V_u, E_u\}$:

- $V_u := \{v_{in}, v_{mid}, v_{out} | v \in V_d\}$. For each of the vertices in the directed graph, we split them into a triplet of in, mid, and out.
- $E_u := \{(u_{out}, v_{in}) | (u,v) \in E_d\} \cup \{(v_{in}, v_{mid}), (v_{mid}, v_{out}) | v \in V_d\}$. 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. ■