1. Morris's counter maintains an integer L using the following algorithms:

   - **Initialize:** At the beginning of the day, set $L \leftarrow 0$.
   - **Update:** Each time a car passes the sensor, increment L with probability 2^{-L}.
   - **Query:** At the end of the day, report $\tilde{n} = 2^L - 1$.

   In the following questions, let $L(n)$ denote the value of L after n updates.

   (a) Prove that $\mathrm{E}[2^{L(n)}] = n + 1$.

   Solution: We prove this claim by induction on n, with the trivial base case $\mathrm{E}[2^{L(0)}] = \mathrm{E}[2^0] = 1$. The key insight is that we can (and arguably *must*) separately consider each possible value of $L(n-1)$.

$$\begin{aligned}
\mathrm{E}[2^{L(n)}] &= \sum_{\ell} \mathrm{E}[2^{L(n)} \mid L(n-1) = \ell] \cdot \Pr[L(n-1) = \ell] && \text{[def. E[· | ·]]} \\
&= \sum_{\ell} \left(\frac{1}{2^\ell} \cdot 2^{\ell+1} + \left(1 - \frac{1}{2^\ell}\right) \cdot 2^\ell \right) \cdot \Pr[L(n-1) = \ell] && \text{[def. } L(n)\text{]} \\
&= \sum_{\ell} (2^\ell + 1) \cdot \Pr[L(n-1) = \ell] && \text{[math]} \\
&= \mathrm{E}[2^{L(n-1)} + 1] && \text{[def. } E[\cdot]\text{]} \\
&= \mathrm{E}[2^{L(n-1)}] + 1 && \text{[linearity]} \\
&= n + 1 && \text{[IH]}
\end{aligned}$$

■

Rubric: 3 points.

Non-solution: Let's try to prove the claim by induction, with the trivial base case $\mathrm{E}[2^{L(0)}] = \mathrm{E}[2^0] = 1$. To simplify notation, I will write $L = L(n)$ and $L^- = L(n-1)$.

$$\begin{aligned}
\mathrm{E}[2^L] &= \mathrm{E}[2^L \mid L = L^-] \cdot \Pr[L = L^-] \;+\; \mathrm{E}[2^L \mid L \neq L^-] \cdot \Pr[L \neq L^-] \\
&= \mathrm{E}[2^{L^-}] \cdot \Pr[L = L^-] \;+\; \mathrm{E}[2^{L^-+1}] \cdot \Pr[L \neq L^-] \\
&= \mathrm{E}[2^{L^-}] \cdot \Pr[L = L^-] \;+\; 2 \cdot \mathrm{E}[2^{L^-}] \cdot \Pr[L \neq L^-] \\
&= n \cdot \Pr[L = L^-] \;+\; 2n \cdot \Pr[L^- \neq L^-] \\
&= n \cdot \left(1 - \frac{1}{2^{L^-}}\right) \;+\; 2n \cdot \frac{1}{2^{L^-}} \qquad \text{(?!)}
\end{aligned}$$

Everything was great until the last step. The problem is that L^- is a random variable, not an actual number! It may be tempting to reason as follows:

$$\begin{aligned}
\cdots &= n \cdot \Pr[L = L^-] \;+\; 2n \cdot \Pr[L \neq L^-] \\
&= n \cdot \left(1 - \frac{1}{\mathrm{E}[2^{L^-}]}\right) \;+\; 2n \cdot \frac{1}{\mathrm{E}[2^{L^-}]} \qquad \text{(?!?!)} \\
&= n \cdot \left(1 - \frac{1}{n}\right) \;+\; 2n \cdot \frac{1}{n} \\
&= n + 1
\end{aligned}$$

This argument is completely bonkers, ***even though it leads to the correct answer.*** You cannot replace random variables with their expectations; the step I've labeled (?!?!) is unjustified. The fact that the argument works out for this particular problem is just a coincidence.

It is possible to prove that $\Pr[L \neq L^-] = 1/n$ using a different argument, but that argument looks like the first solution! ♣

(b) Prove that $\mathrm{E}[4^{L(n)}] = O(n^2)$. *[Hint: Your proof should yield an exact expression, not just a big-O bound.]*

Solution: I claim that $\mathbf{E[4^{L(n)}] = 3n(n+1)/2 + 1}$. We prove this claim by induction on n, with the trivial base case $\mathrm{E}[4^{L(0)}] = \mathrm{E}[4^0] = 1$.

$$\begin{aligned}
\mathrm{E}[4^{L(n)}] &= \sum_\ell \mathrm{E}[4^{L(n)} \mid L(n-1) = \ell] \cdot \Pr[L(n-1) = \ell] && \text{[def. E[$\cdot \mid \cdot$]]} \\
&= \sum_\ell \left(\frac{1}{2^\ell} \cdot 4^{\ell+1} + \left(1 - \frac{1}{2^\ell}\right) \cdot 4^\ell \right) \cdot \Pr[L(n-1) = \ell] && \text{[def. } L(n)\text{]} \\
&= \sum_\ell \left(3 \cdot 2^\ell + 4^\ell\right) \cdot \Pr[L(n-1) = \ell] && \text{[math]} \\
&= \mathrm{E}[3 \cdot 2^{L(n-1)} + 4^{L(n-1)}] && \text{[def. } E[\cdot]\text{]} \\
&= 3 \cdot \mathrm{E}[2^{L(n-1)}] + \mathrm{E}[4^{L(n-1)}] && \text{[linearity]} \\
&= 3n + \mathrm{E}[4^{L(n-1)}] && \text{[part (a)]} \\
&= 3n + \frac{3n(n-1)}{2} + 1 && \text{[IH]} \\
&= \frac{3n(n+1)}{2} + 1 && \text{[math]}
\end{aligned}$$

■

Rubric: 3 points = 1 point for exact closed form + 2 points for proof.

(c) Compute $\mathrm{E}[(\tilde{n} - n)^2]$, where $\tilde{n} = 2^{L(n)} - 1$.

Solution: Definition chasing!

$$\begin{aligned}
\mathrm{E}[(\tilde{n} - n)^2] &= \mathrm{E}[(2^{L(n)} - (n+1))^2] && \text{[def. } \tilde{n}\text{]} \\
&= \mathrm{E}[4^{L(n)} - 2(n+1)2^{L(n)} + (n+1)^2] && \text{[math]} \\
&= \mathrm{E}[4^{L(n)}] \; - \; 2(n+1)\,\mathrm{E}[2^{L(n)}] \; + \; (n+1)^2 && \text{[linearity]} \\
&= \frac{3n(n+1)}{2} + 1 \; - \; 2(n+1)^2 \; + \; (n+1)^2 && \text{[(a) and (b)]} \\
&= \frac{n^2 - n}{2} && \text{[math]}
\end{aligned}$$

■

Rubric: 2 points. 1 point for $O(n^2)$.

(d) Prove that $\Pr[|\tilde{n} - n| > 4n/5] < 4/5$.

Solution: Markov's inequality FTW!

$$\begin{aligned}
\Pr[|\tilde{n} - n| > 4n/5] &= \Pr[(\tilde{n} - n)^2 > (4n/5)^2] \\
&\le \frac{\mathrm{E}[(\tilde{n} - n)^2]}{(4n/5)^2} && \text{[Markov's inequality]} \\
&\le \frac{n^2/2}{(4n/5)^2} && \text{[part (c)]} \\
&= \frac{25}{32} = 0.78125 < \frac{4}{5}.
\end{aligned}$$

■

Rubric: 2 points.

2. Suppose you are given a directed graph $G = (V, E)$, two vertices s and t, a capacity function $c : E \to \mathbb{R}^+$, and a second function $f : E \to \mathbb{R}$. Do not assume ***anything*** about the function f.

   (a) Describe and analyze an efficient algorithm to determine whether f is a maximum (s, t)-flow in G.

   Solution: Verifying that f is a maximum flow requires four steps:

   - Verify that $f(u \to v) \geq 0$ for every edge $u \to v$.
   - Verify that $f(u \to v) \leq c(u \to v)$ for every edge $u \to v$.
   - Verify that $\sum_u f(u \to v) = \sum_w f(v \to w)$ for every node v except s and t.
   - Verify that there is no path from s to t in the residual graph G_f.

   The first three steps can be done in $O(E)$ time by brute force. In the last step, we need $O(E)$ time to build G_f, plus $O(E)$ time to perform a whatever-first search starting at s. Thus, the overall algorithm runs in ***$O(E)$ time***. ■

   Rubric: 5 points = 1 for each step + 1 for time analysis. No penalty for implicitly assuming that flow networks are connected, which implies that $V = O(E)$.

   (b) Describe and analyze an efficient algorithm to determine whether f is the *unique* maximum (s, t)-flow in G.

   Solution: First we verify that f is a maximum (s, t)-flow in G, using the algorithm from part (a). Assume f is a maximum flow, since otherwise we are already done.

   Now we claim that ***f is the unique maximum flow in G if and only if the residual network G_f contains no simple directed cycles of length at least 3.***

   We do need to explicitly rule out cycles of length 2, which correspond to an edges of G that the maximum flow neither saturates nor avoids. The following figure shows a minimal example of a unique maximum flow (on the left) and whose residual network (on the right) contains a cycle of length 2.

   We prove our claim as follows:

   - Suppose the residual graph G_f contains a simple directed cycle $C = v_0 \to v_1 \to \cdots \to v_{\ell-1} \to v_0$ of length $\ell \geq 3$. Because C is simple, all ℓ vertices v_i are distinct, and if C contains the edge $u \to v$, it does not contain the reversed edge $v \to u$. Let $c_{\min} = \min_{e \in C} c_f(e)$ be the minimum residual capacity of any edge in C. Then we can define a new flow $f' = f + c_{\min} \cdot C$

by pushing $c_{\min}$ additional units of flow along C, as follows:

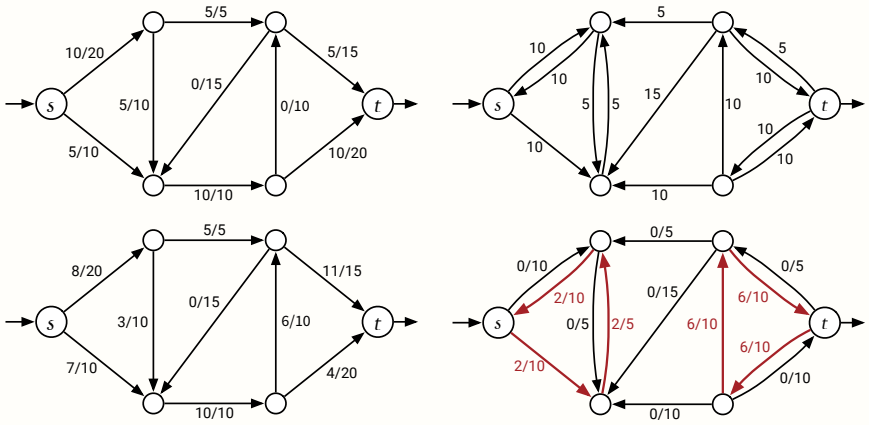
$$f'(x{\to}y) = \begin{cases} f(x{\to}y) + c_{\min} & \text{if } x{\to}y \in C \\ f(x{\to}y) - c_{\min} & \text{if } y{\to}x \in C \\ f(x{\to}y) & \text{otherwise} \end{cases}$$

These cases are exclusive because C is simple. Straightforward definition chasing implies that f' is a feasible flow with the same value as f, and therefore is another maximum flow.

- Suppose there is another maximum (s,t)-flow f'; by definition we have $|f| = |f'|$. We define a flow $g = f' - f$ in the residual graph G_f as follows.

$$g(u{\to}v) = \begin{cases} f'(u{\to}v) - f(u{\to}v) & \text{if } f'(u{\to}v) > f(u{\to}v) \\ f(v{\to}u) - f'(v{\to}u) & \text{if } f'(v{\to}u) < f(v{\to}u) \\ 0 & \text{otherwise} \end{cases}$$

See the example below.



Top left: A maximum flow f. Top right: The residual graph G_f.
Bottom left: Another maximum flow f'. Bottom right: The circulation $g = f' - f$.

Straightforward definition-chasing implies that g is a feasible flow with value 0 in the residual graph G_f. Thus, by the flow decomposition theorem, we can write g as a weighted sum of cycles, each containing only edges where g is positive. None of these cycles use both an edge $u{\to}v$ and its reversal $v{\to}u$, so every cycle has length at least 3.

The simplest method to find nontrivial cycles in the residual graph is essentially brute force. For each edge $u{\to}v$ in the residual graph, we remove the reversed edge $v{\to}u$ from G_f (if it exists), and then look for a path from v to u using whatever-first search. If we find a path from u to v, adding the edge $u{\to}v$ completes the cycle. The resulting algorithm runs in $\boldsymbol{O(E^2)}$ ***time***. ⟨*This is worth 4 points.*⟩

However, there is a faster algorithm that detects nontrivial cycles in $\boldsymbol{O(VE)}$ ***time***. We construct a new directed graph H whose vertices are the *directed edges*

of G_f, and whose edges correspond to a pairs of edges in G_f that can appear consecutively in a simple path. That is, H contains the edge $(u \to v) \to (v \to w)$ if and only if $u \to v \to w$ is a simple directed path in G_f (and in particular $u \neq w$). Overall, the graph H contains $V' = E$ vertices and $E' \leq VE$ edges, and we can construct it in $O(VE)$ time by brute force.

Every closed walk of length ℓ in H corresponds to a closed walk of length ℓ in G_f that does not contain a spur of the form $u \to v \to u$, and vice versa. In particular, G_f contains a simple cycle of length at least 3 if and only if H contains a directed cycle. Equivalently, f is the unique maximum flow if and only if H is a dag. We can check the latter condition in $O(V' + E') = O(VE)$ time using depth-first search. ⟨⟨*This is worth 5 points.*⟩⟩ ■

Solution (+5 extra credit, the harder way): In fact, it is possible to detect nontrivial simple cycles in $O(E)$ *time*. The following algorithm is (morally) due to Donald Johnson [2].

First, we compute the *strong components* of the residual graph G_f, using (for example) Tarjan's algorithm or Kosaraju and Sharir's algorithm, as described in the textbook. Recall that a graph H is strongly connected if every vertex in H can reach every other vertex in H; a strong component of G_f is a maximal strongly connected subgraph. Each edge of G_f belongs to at most one strong component. Each simple cycle in G_f lies entirely inside one strong component, so the rest of our algorithm can search each strong component separately.

Second, we further subdivide the strong components into *biconnected components* at *cut vertices*. A cut vertex is any vertex v whose deletion disconnects the graph; a graph H' is (strongly) biconnected if it is (strongly) connected and has no cut vertices; the biconnected components of a graph H are its maximal biconnected subgraphs. An algorithm of Hopcroft and Tarjan [1] (similar to Tarjan's strong-component algorithm) decomposes any (strongly) connected graph into (strongly) biconnected components in $O(E)$ time. Each nontrivial simple cycle in G_f lies entirely inside one strongly biconnected component, so the rest of our algorithm can search each such component separately.

Finally, let H be any strongly biconnected component of G_f. I claim that **H contains a nontrivial simple cycle if and only if H has at least three vertices.** One direction is trivial—if H has less than three vertices, it cannot contain a cycle with at least three vertices—so let's prove the other. If H has only one or two vertices, the claim is trivial, so assume otherwise. There are two cases to consider.

- Suppose H contains an edge $x \to y$ but not its reversal $y \to x$. Because H is strongly connected, H contains a simple directed path from y to x, which must have length at least 2. Adding the edge $x \to y$ to the path gives us a simple cycle of length at least 3.
- On the other hand, suppose H is symmetric. Fix an arbitrary edge $x \to y$. Because H is biconnected and has at least three vertices, H contains at least two vertex-disjoint paths from y to x; one of these paths is not just the edge

$y \to x$. Adding the edge $x \to y$ to that path gives us a simple cycle of length at least 3.

So the final algorithm looks like this:

```
UniqueMaxFlow(G, f):
  construct the residual graph G_f                    ⟨⟨Ford Fulkerson O(E)⟩⟩
  compute the strong components of G_f                ⟨⟨Tarjan O(E)⟩⟩
  for each strong component H of G_f
      compute the biconnected components of H         ⟨⟨Hopcroft Tarjan O(E_H)⟩⟩
      for each biconnected component H' of H
          if H' has more than two vertices
              return False
  return True
```

The entire algorithm runs in ***O(E) time***. ⟨⟨*This is worth 10 points.*⟩⟩ ■

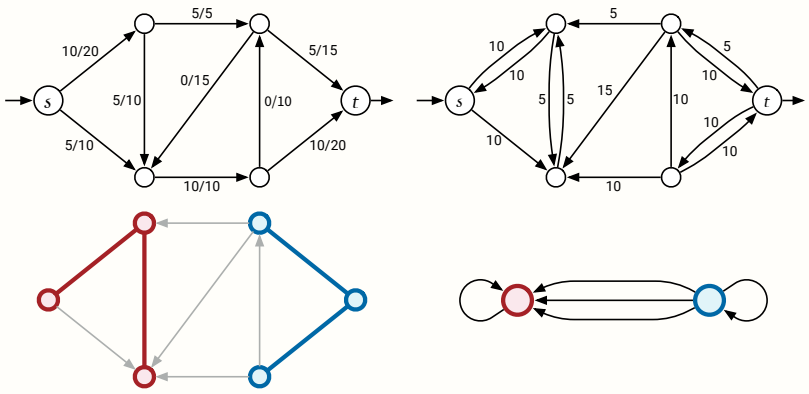
[1] John Hopcroft and Robert E. Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. *Communications of the ACM* 16(6):372–378, 1973.

[2] Donald B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing* 4(1):77-84, 1975.

Solution (+5 extra credit, the easier way): First, let H be the subgraph of G_f induced by all pairs of opposing edges $u \to v$ and $v \to u$. Because H is symmetric, we can treat it as an undirected graph. We can test whether this undirected graph contains a cycle using whatever-first search in $O(E)$ time. If so, then G_f contains a nontrivial simple cycle, so we can report that f is not the only maximum flow.

Otherwise, H is a forest. Let G_f/H denote the (multi-)graph obtained from G_f by *contracting* every edge in H. Each vertex of G_f/H corresponds to either a component of the forest H or an isolated vertex of G_f that is not in H; every edge of G_f/H corresponds to an edge of G_f whose reversal is not in G_f.

I claim that ***G_f contains a nontrivial simple cycle if and only if G_f/H contains a cycle***. In particular, the cycle in G_F/H could have length 1 (a self-loop) or 2. See the figure below for an example.



Top left: A maximum flow f. Top right: The residual graph G_f.
Bottom left: The subgraph H. Bottom right: The contracted graph G_f/H.

As usual we prove this claim in two parts.

⇒ Suppose G_f contains a simple cycle $C = v_1 \to v_2 \to \cdots \to v_\ell \to v_1$, where $\ell \geq 3$. If any edge $v_i \to v_{i+1}$ of C lies in H with its reversal $v_{i+1} \to v_i$, contracting that edge turns C into a shorter simple cycle. Contracting a symmetric edge pair $v_i \leftrightarrow v_j$ between two non-adjacent vertices turns C into a figure-8: two simple cycles sharing a single vertex. In both cases, the induction hypothesis implies that contracting the remaining edges of H leaves a multigraph with at least one cycle.

⇐ Suppose G_f/H contains a simple cycle C, possibly of length 1 or 2. Each edge of C corresponds to an edge in G_f. Let $u_0 \to v_1, u_1 \to v_2, \ldots, u_k \to v_0$ denote the edges of G_f corresponding to the edges of C in order. For each index i, vertices u_i and v_i lie in the same component of H, and thus are connected by a unique path in H. Because C is a simple cycle, it touches each component of H at most once, so the various paths in H are disjoint. We conclude that connecting the edges $u_{i-1} \to v_i$ with the paths $v_i \rightsquigarrow u_i$ through H yields a simple cycle in G_f that never uses both an edge and its reversal.

We can construct G_f/H in $O(E)$ time by brute force, and then we can determine whether G_f/H is a directed acyclic graph in $O(E)$ time using (for example) depth-first search. The overall algorithm runs in ***O(E) time***. ⟨⟨*This is worth 10 points.*⟩⟩ ■

Rubric: 5 points = 2 points for "G_f has no nontrivial simple cycles" + 2 points for algorithm + 1 for running time. Proof of correctness is not required. Scale partial credit as follows:

- Max 4 points for $O(E^2)$ time.
- Max 5 points for $O(VE)$ time.
- Max 10 points(!) for $O(E)$ time.
- Only 1 point for "G_f is a dag".

3. Suppose you are given an $n \times n$ checkerboard with some of the squares deleted. You have a large set of dominos, just the right size to cover two squares of the checkerboard. Describe and analyze an algorithm to determine whether one tile the board with dominos—each domino must cover exactly two undeleted squares, and each undeleted square must be covered by exactly one domino.

Solution (Reduction to bipartite perfect matching): Given the array $Deleted[1..n, 1..n]$, we first construct a bipartite graph $G = (L \cup R, E)$ as follows:

- $L := \{(i,j) \mid Deleted[i,j] = \text{FALSE} \text{ and } i+j \text{ is even}\}$ — Intuitively, L is the set of undeleted white squares.
- $R := \{(i,j) \mid Deleted[i,j] = \text{FALSE} \text{ and } i+j \text{ is odd}\}$ — Intuitively, R is the set of undeleted black squares.
- $E := \{(i,j)(i',j') \mid (i,j) \in L \text{ and } (i',j') \in R \text{ and } |i-i'|+|j-j'| = 1\}$ — Intuitively, E is the set of all adjacent pairs of undeleted squares.

Every domino must cover one white square and one black square, so every matching in G represents a valid placement of dominos and vice versa. We compute a ***maximum-cardinality matching*** in G; if this matching is perfect, we return TRUE, and otherwise, we return FALSE. If we use the matching algorithm presented in class, our algorithm runs in $O(VE) = \boldsymbol{O(n^4)}$ ***time.*** ■

Solution (Reduction to maximum flow): Given the array $Deleted[1..n, 1..n]$, we first construct a flow network $G = (V, E)$ as follows:

- $V := \{s,t\} \cup \{(i,j) \mid Deleted[i,j] = \text{FALSE}\}$.
- There are three types of edges:
  - $s \to (i,j)$ for all $(i,j) \in V$ where $i+j$ is even.
  - $(i,j) \to t$ for all $(i,j) \in V$ where $i+j$ is odd.
  - $(i,j) \to (i',j')$ for all $(i,j), (i',j') \in V$ where $i+j$ is even and $|i-i'|+|j-j'| = 1$

  Intuitively, we have an edge from s to each white square, from each white square to each adjacent black square, and from each adjacent black square to t.
- Every edge in G has capacity 1.

Now we compute a maximum (s,t)-flow in G. If the value of this flow is exactly half the number of undeleted squares, return TRUE; otherwise, return FALSE.

The maximum flow value is equal to the largest number of non-overlapping dominos we can place on the board, which is trivially at most $n^2/2$. Thus, if we use Ford-Fulkerson to compute the maximum flow, our algorithm runs in $O(n^2E) = \boldsymbol{O(n^4)}$ ***time.*** If we use Orlin's algorithm to compute the maximum flow, our algorithm still runs in $O(VE) = \boldsymbol{O(n^4)}$ ***time.*** ■

Rubric: 10 points: standard graph-reduction rubric (see next page)

Standard graph-reduction rubric. For problems solved by reduction to a standard graph algorithm covered either in class or in a prerequisite class (for example: shortest paths, topological sort, minimum spanning trees, maximum flows, bipartite maximum matching, vertex-disjoint paths, ...).

Maximum 10 points =

- \+ 3 point for defining the **correct graph**.
  - \+ 1 for correct vertices
  - \+ 1 for correct edges
  - — ½ for forgetting "directed" if the graph is directed
  - \+ 1 for correct data associated with vertices and/or edges—for example, weights, lengths, capacities, costs, demands, and/or labels—if any
  - ○ The vertices, edges, and associated data (if any) must be described as explicit functions of the input data.
- ○ For most problems, the correct graph can be **constructed** in linear time by brute force; in this common case, no explicit description of the construction algorithm is required. If achieving the target running time requires a more complex algorithm, that algorithm will graded out of 5 points using the appropriate standard rubric, and all other points are cut in half.
- \+ 2 points for explicitly relating the given problem to a specific **problem** involving the constructed graph. For example: "The minimum number of moves is equal to the length of the shortest path in G from $(0, 0, 0)$ to any vertex of the form $(k, \cdot, \cdot)$ or $(\cdot, k, \cdot)$ or $(\cdot, \cdot, k)$." or "Each path from s to t represents a valid choice of class, room, time, and proctor for one final exam; thus, we need to construct a path decomposition of a maximum (s, t)-flow in G."
  - ○ No points for just writing (for example) "shortest path" or "reachability" or "matching". Shortest path in which graph, from which vertex to which other vertex? How does that shortest path relate to the original problem?
  - ○ No points for only naming the algorithm, not the problem. "Breadth-first search" and "Ford-Fulkerson" are not problems!
- \+ 3 points for correctly applying the correct black-box **algorithm** to solve the stated problem. (For example, "Perform a single breadth-first search in H from $(0, 0, 0)$ and then examine every target vertex." or "We compute the maximum flow using Ford-Fulkerson, and then decompose the flow into paths as described in the textbook.")
  - ○ This includes algorithmic details of extracting the correct answer to the stated problem from the output pf the black-box algorithm.
  - —1 for *explaining* an algorithm from lecture or the textbook instead of just invoking it as a black box.
- \+ 2 points for correctly stating the running time in terms of the *input* parameters (not just the number of vertices and edges of the constructed graph).

An extremely common mistake for this type of problem is to attempt to modify a standard algorithm and apply that modification to the input data, instead of modifying the input data and invoking a standard algorithm as a black box. This strategy can work in principle, but it is much harder to do it correctly, and it is terrible software engineering practice. ***Clearly correct*** solutions using this strategy will be given full credit, but partial credit will be given only sparingly.