1. Let $G = (V, E)$ be a directed graph with non-negative edge lengths $\ell(e), e \in E$ that represents a road network. Alice is invited to a party at her friend Bob's house and she wants to buy dessert at a grocery store on the drive to his house. Just as she is about to leave she realizes that she may not have sufficient gas in her car to drive all the way to Bob's house. Her car can go at most a distance of R before the gas runs out. Describe an efficient algorithm to help Alice accomplish the task of reaching Bob's house with as little travel as feasible; she needs to buy dessert but may or may not need to fill gap. Assume Alice's house is at node s and Bob's house is at node t and that the grocery shops are given by a set $X \subset V$ and the gas stations by a set $Y \subset V$. Assume that X, Y are disjoint sets. Also assume, for simplicity, that once Alice fills gas she can travel an infinite distance. Note that Alice could buy dessert either before or after filling up gas, as long as she does not run out of fuel on the way to the gas station. Express your running time as a function of n, the number of nodes, and m, the number of edges.

Solution (Graph Modeling): The high-level idea is to construct four copies of G, corresponding to: having visited neither a gas station nor a grocery store, having visited a gas station but not a grocery store, having visited a grocery store but not a gas station, and having visited both. We add length 0 edges between relevant copies of gas stations and grocery stores. To account for the maximum possible distance R, we will discard all vertices whose distance from s is greater than R in the copies that represent not having visited a gas station yet.

- Let $V_{\text{close}} = \{v \in V \mid dist(s, v) \leq R\}$. As explained above, we set
$$V' := \big(V_{\text{close}} \times \{\mathsf{None}, \mathsf{Grocery}\}\big) \cup \big(V \times \{\mathsf{Gas}, \mathsf{Both}\}\big)$$
as the vertex set.
- We will form our edge set as the union of three sets:
  - $E_0 := \{(u, a) \to (v, a) \mid u \to v \in E, a \in \{\mathsf{None}, \mathsf{Grocery}, \mathsf{Gas}, \mathsf{Both}\}\}$, copying the edges that existed in the original graph G. For the edges in E_0 we set $\ell'((u, a) \to (v, a)) = \ell(u \to v)$.
  - $E_1 := \{(x, \mathsf{None}) \to (x, \mathsf{Grocery}) \mid x \in X\} \cup \{(x, \mathsf{Gas}) \to (x, \mathsf{Both}) \mid x \in X\}$, allowing us to move from a copy where we haven't visited a grocery store to a copy where we have. For edge $e \in E_1$, $\ell'(e) = 0$.
  - $E_2 := \{(y, \mathsf{None}) \to (y, \mathsf{Gas}) \mid y \in Y\} \cup \{(y, \mathsf{Grocery}) \to (x, \mathsf{Both}) \mid y \in Y\}$, allowing us to move from a copy where we haven't visited a gas station to a copy where we have. For edge $e \in E_2$, $\ell'(e) = 0$.

  In summary, $E' := E_0 \cup E_1 \cup E_2$.

  As made evident by the notation, all edges are directed.
- There are two possibilities: there is a path from s to a grocery store to t via a path of length at most R, or every path from s to a grocery store to t has length greater than R.

  In the former case, we want the shortest path from (s, None) to $(t, \mathsf{Grocery})$; in the latter, we want the shortest path from (s, None) to (t, Both).

  We can determine which case is correct and return the corresponding path by solving the *single-source shortest paths problem* from (s, None).

- The edge lengths are non-negative, we can compute the shortest paths from (s, None) using Dijkstra's algorithm. The graph is not necessarily a DAG, so we cannot necessarily use Dynamic Programming.
- We can compute V_{close} by running Dijkstra's algorithm on G from s and taking all vertices whose distance from s is at most R. Once that is done, G' can be computed via brute force in $O(V' + E') = O(V + E)$ time, since G' has at most four times as many vertices/edges as G. Running Dijkstra's algorithm on G' takes $O(E' \log V') = O(E \log V)$ time using a binary heap, or $O(E' + V' \log V') =$ $\boldsymbol{O(E + V \log V)}$ ***time*** using a Fibonacci heap.

■

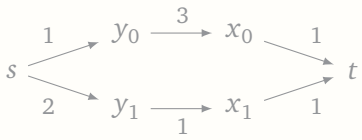
Solution (Combining multiple shortest path computations): The possibilities for the optimal solution are as follows:

1. The path goes from s to a grocery store to t via a path of length at most R.
2. Every path from s to t via a grocery store has length greater than R, so either
   a. the path from s to t passes through a gas station and then a grocery store, or
   b. the path from s to t passes through a grocery store and then a gas station.

To check case 1, we want the minimum of $dist(s, x) + dist(x, t)$ over all $x \in X$ such that $dist(s, x) + dist(x, t) \leq R$. Computing $dist(s, v)$ for $v \in V$ can be done by running Dijkstra's algorithm on G from s. A naïve approach to computing $dist(x, t)$ would be running Dijkstra's algorithm $|X|$ times, once from each $x \in X$, but one can do better. Notice that for $v \in V$, $dist(v, t)$ is the same as the distance from t to v in G^{rev}, so we can compute $dist(v, t)$ for $v \in V$ by running Dijkstra's algorithm *once* on G^{rev} from t. We can then iterate over all $x \in X$ and check if the minimum is at most R.

In case 2, we want to try both subcases and take the shorter of the two paths. We will investigate each case in turn.

In case 2a, we want the minimum of $dist(s, y) + dist(y, x) + dist(x, t)$ over all $x \in X$ and $y \in Y$ such that $dist(s, y) \leq R$. A naïve approach for computing the relevant values of $dist(s, y) + dist(y, x) + dist(x, t)$ would be to run Dijkstra's algorithm from each $y \in Y$ such that $dist(s, y) \leq R$, and then run Dijkstra's algorithm from each $x \in X$. We can optimize the last step as before by replacing it with a single call to Dijkstra's algorithm on G^{rev} from t (or just reuse the computation from the previous case), but in fact we can do even better than that by observing we only care about *shortest* paths. Create a new vertex s^*, and add edges $s^* \to y$ for each $y \in Y$ such that $dist(s, y) \leq R$, with $\ell(s^*, y) = dist(s, y)$. Then the shortest path of the form $s \rightsquigarrow y \rightsquigarrow x$ where $y \in Y$ with $dist(s, y) \leq R$ and $x \in X$ corresponds directly to the shortest path $s^* \to y \rightsquigarrow x$, so we can use $dist(s^*, x)$ in place of $dist(s, y) + dist(y, x)$ when finding the minimum. We already have $dist(x, t)$ for all $x \in X$ from case 1, so we can find the minimum over all such paths by simply iterating over $x \in X$. It is tempting to try to *greedily* find the shortest path of the form $s \rightsquigarrow y \rightsquigarrow x \rightsquigarrow t$ as follows: first find $y_0 = \arg\min_{y \in Y} dist(s, y)$, then compute $x_0 = \arg\min_{x \in X} dist(y, x)$, and then output the path $s \rightsquigarrow y_0 \rightsquigarrow x_0 \rightsquigarrow t$. This is incorrect, as can be seen in the following example.



Set $X = \{x_0, x_1\}$ and $Y = \{y_0, y_1\}$. The greedy approach would select the path $s \to y_0 \to x_0 \to t$ which has total length 5 instead of the path $s \to y_1 \to x_1 \to t$ which has total length 4. The difference between this greedy approach and the (correct) approach described above is somewhat subtle.

Finally, we come to case 2b. We want the minimum of $dist(s,x) + dist(x,y) + dist(y,t)$ over all $x \in X$ and $y \in Y$ such that $dist(s,x) + dist(x,y) \le R$. Naïvely, we would compute $dist(x,y)$ for all $x \in X$ and $y \in Y$. This can be done by either running Dijkstra's algorithm from each $x \in X$, or from each $y \in Y$ in G^{rev} for a total of $\min\{|X|, |Y|\}$ calls to Dijkstra's algorithm. Once again, we can do better than that. We can apply the trick from case 2a as follows. Add a vertex $s^\dagger$ with edges $s^\dagger \to x$ for all $x \in X$, of lengths $\ell(s^\dagger \to x) = dist(s,x)$. Then the shortest path of the form $s \rightsquigarrow x \rightsquigarrow y$ where $x \in X$ corresponds directly to the shortest path $s^\dagger \to x \rightsquigarrow y$. As before, we use $dist(s^\dagger, y)$ in place of $dist(s,x) + dist(x,y)$, and once again, we already have $dist(y,t)$ for all $y \in Y$ from case 1, so we just need to iterate over all y such that $dist(s^\dagger, y) \le R$.

Streamlining some of the constructions and computations above, the algorithm can be summarized as follows:

BringDessertToParty(G, X, Y, s, t, R):
- Run Dijkstra's algorithm on G from s and record $dist(s,v)$ for all $v \in V$
- Run Dijkstra's algorithm on G^{rev} from t and record $dist(v,t)$ for all $v \in V$
- Find $\min_{x \in X} \{dist(s,x) + dist(x,t)\}$. If this value is at most R, return it. Otherwise:
  - Create graph G' from G by adding vertices s^* and $s^\dagger$, with edges $s^* \to y$ of length $\ell(s^* \to y) = dist(s,y)$ for $y \in Y$ where $dist(s,y) \le R$, and edges $s^\dagger \to x$ of length $\ell(s^\dagger \to x) = dist(s,x)$ for $x \in X$.
  - Run Dijkstra's algorithm on G' from s^* and record $dist(s^*, x)$ for $x \in X$.
  - Run Dijkstra's algorithm on G' from $s^\dagger$ and record $dist(s^\dagger, y)$ for $y \in Y$.
  - Compute and return the smaller of $\min_{x \in X} \{dist(s^*, x) + dist(x,t)\}$ and $\min_{y \in Y, dist(s^\dagger, y) \le R} \{dist(s^\dagger, y) + dist(y,t)\}$.

Computing G^{rev} and G' each take $O(V+E)$ time, and the various min calculations each take $O(V)$ time. Thus the running time of this algorithm is dominated by the (at most) four calls to Dijkstra's algorithm, giving $\boldsymbol{O(E + V \log V)}$ ***time*** overall. ■

Rubric:

- For full points, the solution needs to have the same asymptotic running time as Dijkstra's algorithm (i.e., at most a constant number of calls to Dijkstra's algorithm). No penalty for citing the running time of Dijkstra's algorithm as $O(E \log V)$.
- 8 points max for an $O(VE \log V)$ time algorithm (e.g., overly large graph construction if using graph reduction, or making $O(V)$ calls to Dijkstra's algorithm if combining multiple shortest path computations).
- 6 points max for a polynomial time algorithm slower than $O(VE \log V)$.
- For a graph reduction solution, use the Standard graph reduction rubric (see last page).
- For a combining multiple shortest path computations solution:
  - 2 points for correct case analysis
  - 6 points for the correct computations in each case. Do not penalize here for inefficiencies here, only for correctness.
    - -1 for each minor error
    - -2 for each major error
  - 1 point for putting the computations from the various cases together
  - 1 point for time analysis in terms of the input parameters

2. Let $G = (V, E)$ be a directed graph with non-negative edge lengths $\ell(e), e \in E$. Dijkstra's algorithm can be used to find the shortest path tree rooted at any given node $s \in V$. In the standard shortest path problem the length of a path $v_1, v_2, \ldots, v_h$ is defined as $\sum_{i=1}^{h-1} \ell(v_i, v_{i+1})$ which is simply the sum of the lengths of the edges in the path. In various situations one needs different measures.

   - For a parameter $k \geq 1$ the k-norm length of a path $v_1, v_2, \ldots, v_k$ is defined to be $(\sum_{i=1}^{h-1} \ell(v_i, v_{i+1})^k)^{1/k}$. If $k = 1$ we get the standard length. Given G, $s, t \in V$ and $k \geq 1$ describe an algorithm to find the shortest k-norm length path from s to t in G. Give a small example where 2-norm s-t shortest path is different from the standard shortest path.

Solution (Reweighting the graph): Suppose $v_1, \ldots, v_h$ is the shortest path from v_1 to v_h. Let $kd(v_1, v_h)$ denote the k-norm length of this path, i.e., the k-norm distance between v_1 and v_h. Note that for the sake of comparing lengths, taking k-th roots is superfluous: for vertices u, v, x, y, $kd(u, v) \leq kd(x, y)$ if and only if $kd(u, v)^k \leq kd(x, y)^k$. Furthermore, we observe directly that $kd(v_1, v_h)^k = \sum_{i=1}^{h-1} \ell(v_i \to v_{i+1})^k$.

Thus if we set a new edge length $\ell'(u \to v) := \ell(u \to v)^k$ for each edge $u \to v$, and let $d'(u, v)$ be the (standard) distance between u and v with respect to the new lengths ℓ', $d'(u, v)$ is *exactly* $kd(u, v)^k$. Thus running Dijkstra's algorithm from s with the new edge lengths results in finding the shortest k-norm length path from s to every other vertex.

Figuring out the running time of this algorithm can be somewhat tricky. Note that the number of bits needed to write down $\ell'(u \to v)$ is at most k times the number of bits needed to write down $\ell(u \to v)$, so that if k is treated as a constant, then the asymptotic running time is the same as that of Dijkstra's algorithm. However, if k is not constant then we would need to be more careful.

Note that while the reduction produces the correct shortest k-norm length path, the algorithm will report the length as $kd(s, t)^k$ instead of $kd(s, t)$. To obtain the correct k-norm length, we would need to take k-th roots; however, in general k-th roots are irrational, meaning that we would get into questions of representation, numerical precision and the associated computational complexity of such matters.

In the graph below, the x-z shortest path is $x \to z$ with length 6, whereas the 2-norm x-y shortest path is $x \to y \to z$ with 2-norm length $\sqrt{3^2 + 4^2} = 5$. We can also observe the reweighted graph producing the correct shortest k-norm length path.

■

Solution (Modifying Dijkstra's algorithm): Recall that Dijkstra's algorithm works by maintaining a guess $dist(s,v)$ of the *true* distance $d(s,v)$ from s to v for each vertex $v \in V$, with the invariant that $dist(s,v) \geq d(s,v)$, and updates $dist(s,v)$ via the rule $dist(s,v) \leftarrow \min\{dist(s,v), dist(s,u) + \ell(u \to v)\}$. In the textbook, this is referred to as *relaxing* the edge $u \to v$. This relaxation rule is based on the fact that if the shortest path from s to v ends in the edge $u \to v$, then the true distance $d(s,v)$ is *exactly* $d(s,u) + \ell(u \to v)$.

We can modify the relaxation rule to work for k-norm lengths. If $kd(s,v)$ were the *true* k-norm distance from s to v, the analysis from the previous solution states that $kd(s,v)^k = kd(s,u)^k + \ell(u \to v)^k$. In summary, if we maintain a guess $kdist^k(s,v)$ of $kd(s,v)^k$, when updating we should use the relaxation rule $\boldsymbol{kdist^k(s,v) \leftarrow \min\left\{kdist^k(s,v), kdist^k(s,u) + \ell(u \to v)^k\right\}}$.

Note that the resulting behavior is *exactly* the same as running the standard Dijkstra's algorithm on the reweighted graph in the previous solution. Accordingly, we have the same considerations about the bit complexity needed to write down the guesses $kdist^k(s,v)$. If k is a constant, then the asymptotic running time is the same as that of the standard version of Dijkstra's algorithm.

One may wonder why we keep a guess of $kd(s,v)^k$ instead of $kd(s,v)$. Trying to maintain a guess for $kd(s,v)$ would require taking a k-th root every time we relax an edge. As with the previous solution, that gets us into tricky matters about representation, numerical precision, etc.

The modified pseudocode is shown below, following the presentation of Dijkstra's algorithm from the lecture slides. We use GetVal(Q,u) as a routine that given a key u in a priority queue Q return the value stored in Q for u. Modifications are shown in green.

```
kNormDijkstra(G, s):
  Q ← MakePQ()
  Insert(Q, (s, 0))
  for each vertex v ≠ s
      Insert(Q, (v, ∞))
  X ← ∅
  for i from 1 to |V|
      (u, kdist^k(s,u)) ← ExtractMin(Q)
      X ← X ∪ {u}
      for each edge u→v
          if (v ∉ X)
              DecreaseKey(Q, min{GetVal(Q, v), kdist^k(s,u) + ℓ(v_h→v_{h+1})^k})
```

■

- Consider the previous part but now suppose we set k to be a very large number. As $k \to \infty$ the k-norm of a path can be seen to be the maximum length of the edges in the path (assume that edge lengths are distinct). This corresponds to the ∞ norm of a vector which is the largest coordinate. In the context of paths, the length of the longest edge length in a path is called its *bottleneck* length. Describe an algorithm to compute the bottleneck shortest path distances from s to every node in G by adapting Dijkstra's algorithm.

Solution: We adapt the analysis from the second solution for the previous part. For vertices u and v, let $b(u,v)$ be the *true* bottleneck distance between u and v. Suppose the bottleneck shortest path from s to v ends in the edge $u \to v$. Then $b(s,v) = \max\{b(s,u), \ell(u \to v)\}$. Thus if we maintain a guess $bottle(s,v)$, we relax via $\boldsymbol{bottle(s,v) \leftarrow \min\left\{bottle(s,v), \max\{bottle(s,u), \ell(u \to v)\}\right\}}$.

As with the previous part, the asymptotic running time is the same as that of the standard version of Dijkstra's algorithm.

For completeness we include the pseudocode (modifications are in green):

```
BottleneckDijkstra(G, s):
  Q ← MakePQ()
  Insert(Q, (s, 0))
  for each vertex v ≠ s
      Insert(Q, (v, ∞))
  X ← ∅
  for i from 1 to |V|
      (u, bottle(s,u)) ← ExtractMin(Q)
      X ← X ∪ {u}
      for each edge u→v
          if (v ∉ X)
              DecreaseKey(Q, min{GetVal(Q, v), max{bottle(s,u), ℓ(u→v)}})
```

■

Rubric:

- 6 points for the first part:
  - 5 points for a correct algorithm:
    - Scaled graph reduction rubric (see last page) if appropriate
    - Otherwise: 4 points for the algorithm, 1 point for the time analysis; −1 for a minor error, −2 for each major error.
  - 1 points for a correct example.
- 4 points for the second part:
  - 3 points for a correct algorithm, 1 point for time analysis.
  - −1 for each minor error.

3. Since you are taking an algorithms class you decided to create a fun candy hunting game for Halloween. You set up a maze with one way streets that can be thought of as a directed graph $G = (V, E)$. Each node v in the maze has $w(v)$ amount of candy located at v.

   Before you ask your friends to solve the game you need to know how to do it yourself! Describe efficient algorithms for both variants. Ideally your algorithm should run in linear time.

   - Each of your friends, starting at a given node s, has to figure out the maximum amount of candy they can collect. Note that candy at node v can be collected only once even if the node v is visited again on the way to some other place.

Solution (Pure Graph Modeling): We will first solve the problem in two special cases: when G is strongly connected, and when G is a DAG. We will then put the two pieces together into a single algorithm for the general case.

- Suppose G is strongly-connected. Then starting from s one can visit every other vertex, so the maximum amount of candy is simply $\sum_{v \in V} w(v)$.
- Suppose G is a DAG. We will reduce to a single source-shortest paths problem, so that we can invoke the linear-time Dynamic Programming based algorithm for DAGs seen in class.
  The single-source shortest paths problem involves *edge lengths* and not *vertex weights*, so we will use a "vertex-splitting" trick to convert the vertex weights into edge lengths: for each vertex $v \in V$, we will create two vertices v^- and v^+, so that each edge $u \to v \in E$ becomes an edge $u^+ \to v^-$, and then we will add edges $v^- \to v^+$ that carries the weight of v as its length. Note that single-source shortest paths is a *minimization* problem; here we are trying to solve a *maximization* problem, so we will set $\ell(v^- \to v^+) = -w(v)$.
  * $V' := \left\{ v^-, v^+ \;\middle|\; v \in V \right\}$
  * $E' := \left\{ u^+ \to v^- \;\middle|\; u \to v \in E \right\} \cup \left\{ v^- \to v^+ \;\middle|\; v \in V \right\}$, where $\ell(u^+ \to v^-) := 0$ and $\ell(v^- \to v^+) := -w(v)$.
  * We need to solve the *single-source shortest paths* problem from s^-, and then find the vertex $v_0 = \arg\min_{v \in V'} dist(s^-, v)$. Then the maximum amount of candy will be $-dist(s^-, v_0)$.
  * Since G is a DAG, we will run the linear-time Dynamic Programming based single-source shortest paths algorithm for DAGs seen in class.
  * Building the new graph $G' = (V', E')$ takes $O(V' + E') = O(V + E)$ time to build by brute force. Then running the Dynamic Programming based single-source shortest paths algorithm for DAGs takes $O(V' + E') = O(V + E)$ time, for a total of $O(V + E)$ time.

In the general case, upon entering a strongly-connected component, we can temporarily pretend we are in the case where G is strongly-connected, and walk around collecting all the candy in the component before leaving.

Thus we can imagine operating on the strongly-connected component meta-graph G^{SCC}, where every time we enter a metavertex $\bar{v}$ of G^{SCC}, we collect all the candy in the component $\mathsf{comp}(\bar{v})$ in G corresponding to $\bar{v}$ before moving on. As a result, we should set $w(\bar{v}) = \sum_{u \in \mathsf{comp}(\bar{v})} w(u)$. Since G^{SCC} is a DAG, we can

compute the maximum amount of candy that can be collected by performing the reduction to SSSP described above on G^{SCC} with the metavertex $\bar{s}$ corresponding to the strongly-connected component containing s.

Computing G^{SCC} takes $O(V+E)$ time using the algorithm described in lecture, and solving the problem on $G^{SCC} = (V^{SCC}, E^{SCC})$ takes takes $O(V^{SCC} + E^{SCC}) = O(V+E)$ time. Overall the running time is $O(V+E)$, as desired. ■

Solution (Dynamic Programming for DAG case): We provide an alternative solution for the case when G is a DAG: instead of reducing to single-source shortest paths via graph modeling, we will design a custom dynamic programming solution. Instead of applying the vertex-splitting trick to G^{SCC} in the previous solution, we can run *MaxCandyDAG* (described below) on G^{SCC} and $\bar{s}$ instead.

Let $v_1, \ldots, v_n$ be a topological ordering of the vertices of G. For $1 \le i \le n$, let $MaxCandy(i)$ denote the maximum amount of candy one can collect starting at vertex v_i. If i_s is the index for vertex s, we should return $\boldsymbol{MaxCandy(i_s)}$. The function obeys the recurrence

$$MaxCandy(i) = w(v_i) + \max_{v_i \to v_j \in E} \left\{ MaxCandy(j) \right\}.$$

Note that when v_i is a sink, we are taking the max over an empty set, which we interpret to be 0, so no explicit base case is necessary. Since $MaxCandy(i)$ only depends on $MaxCandy(j)$ where $i < j$ (due to the topolgoical ordering), we can memoize this recurrence into a one-dimensional array, in order from n down to 1 (i.e., in *reverse topological order*).

```
MaxCandyDAG(G,s):
  v_1,...,v_n ← a topological ordering of the vertices of G
  for i ← n down to 1
      bestEdgeCandy ← 0
      for each edge v_i→v_j
          bestEdgeCandy ← max{bestEdgeCandy, MaxCandy[j]}
      MaxCandy[i] ← w(v_i) + bestEdgeCandy
  return MaxCandy[i_s] where i_s is the index of s
```

In spite of the double for-loop, the inner for-loop executes exactly once per edge, and so the running time of this algorithm is $O(V+E)$.

An aside about recursion and DAGs: since DAGs have no cycles, it is valid to consider recursion on the DAG itself, just like in the case of trees. Thus we could have declared that for vertex u, $MaxCandy(u)$ denotes the maximum amount of candy one can collect starting at vertex v, and stated that we should return $MaxCandy(s)$. In this setting,

$$MaxCandy(u) = w(u) + \max_{u \to v \in E} \left\{ MaxCandy(v) \right\}$$

is a well-defined recursive formulation for the function. Note that we are implicitly relying on the fact that this graph is acyclic–we cannot do this for arbitrary directed graphs that have cycles in them. Nevertheless, formulating the recurrence this way does not change the fact that the correct evaluation order is *reverse topological order* (i.e., a post-order traversal), so the pseudocode version would be the same. ■

- Your friends complain that they can collect more candy if they get to choose the starting node. You agree to their their request and ask them to maximize the amount of candy they can collect starting at any node they choose.

Solution (Graph Modeling): We can simulate choosing an arbitrary starting vertex by adding a new vertex s^* with $w(s^*) = 0$, along with an edge $s^* \to v$ for each $v \in V$. One can think of this as all of the friends gathering together (at s^*) before going off to their chosen vertices.

We then run either algorithm from the previous part on this new graph with s^* as the new starting vertex. The new graph has has $V + 1$ vertices and $V + E$ edges, so the asymptotic running time does not change. ■

Solution (Modify the Dynamic Programming solution): Since the algorithm in the previous part uses a reduction to the case when G is a DAG, it suffices to look at modifying that special case. Specifically, if using the custom Dynamic Programming solution, we can make the following minor modification: instead of specifying s and returning $MaxCandy(i_s)$, we return $\mathbf{max_{1 \le i \le n}\, MaxCandy(i)}$.

```
MaxCandyDAGWithChoice(G):
  v_1, ..., v_n ← a topological ordering of the vertices of G
  maxOverall ← 0
  for i ← n down to 1
      bestEdgeCandy ← 0
      for each edge v_i → v_j
          bestEdgeCandy ← max{bestEdgeCandy, MaxCandy[j]}
      MaxCandy[i] ← w(v_i) + bestEdgeCandy
      maxOverall ← max{maxOverall, MaxCandy[i]}
  return maxOverall
```

The asymptotic running time of the algorithm does not change. ■

Rubric:

- 8 points for the first part. 5 points for a correct algorithm that is slower than linear time.
  - For reductions to G^{SCC}:
    - 1 points for the case where G is strongly-connected (i.e., the weights on the metavertices)
    - 5 points for the case where G is a DAG: scaled Dynamic Programming rubric or Graph reduction rubric (see last page), as appropriate.
    - 2 point for putting the two pieces together. −1 for a minor error.
  - Otherwise: −1 for a minor errors, −2 for major errors.
- 2 points for the second part. −1 for a minor error.

Rubric (Standard rubric for graph reduction problems): For problems out of 10 points:

- \+ 1 for correct vertices, *including English explanation for each vertex*
- \+ 1 for correct edges
  - — ½ for forgetting "directed" if the graph is directed
- \+ 1 for stating the correct problem
  - — "Breadth-first search" is not a problem; it's an algorithm!
- \+ 1 for correctly applying the correct algorithm
  - — ½ for using a slower or more specific algorithm than necessary
- \+ 1 for time analysis in terms of the input parameters.
- \+ 5 for other details of the reduction
  - — If your graph is constructed by naive brute force, you do not need to describe the construction algorithm; in this case, points for vertices, edges, problem, algorithm, and running time are all doubled.
  - — Otherwise, apply the appropriate rubric, *including Deadly Sins*, to the construction algorithm. For example, for a solution that uses dynamic programming to build the graph quickly, apply the standard dynamic programming rubric.

Rubric (Standard dynamic programming rubric): For problems worth 10 points:

- 6 points for a correct recurrence, described either using mathematical notation or as pseudocode for a recursive algorithm.
  - \+ 1 point for a clear **English** description of the function you are trying to evaluate. (Otherwise, we don't even know what you're *trying* to do.) **Automatic zero if the English description is missing.**
  - \+ 1 point for stating how to call your function to get the final answer.
  - \+ 1 point for base case(s). −½ for one *minor* bug, like a typo or an off-by-one error.
  - \+ 3 points for recursive case(s). −1 for each *minor* bug, like a typo or an off-by-one error. **No credit for the rest of the problem if the recursive case(s) are incorrect.**
- 4 points for details of the dynamic programming algorithm
  - \+ 1 point for describing the memoization data structure
  - \+ 2 points for describing a correct evaluation order; a clear picture is usually sufficient. If you use nested loops, be sure to specify the nesting order.
  - \+ 1 point for time analysis
- Unless otherwise specified, it is *not* necessary to state a space bound.
- For problems that ask for an algorithm that computes an optimal *structure*—such as a subset, partition, subsequence, or tree—an algorithm that computes only the *value* or *cost* of the optimal structure is sufficient for full credit, unless the problem says otherwise.
- Official solutions usually include pseudocode for the final iterative dynamic programming algorithm, ***but iterative pseudocode is not required for full credit***. If your solution includes iterative pseudocode, you do not need to separately describe the recurrence, memoization structure, or evaluation order. (But you still need to describe the underlying recursive function in English.)