1. Several problems can be solved on directed graphs by first solving the corresponding problem on a DAG, and then reducuing the problem on a general graph to that on a DAG via the meta-graph. On a DAG one can use topological sort or some other useful property. Solve each of the following problems using this general paradigm.

   (a) (3 pts) Let $G = (V, E)$ be a directed graph such that each vertex v has a number $b(v)$ associated with it. The goal is to compute for each $u \in V$ the number $a(u) = \min\{b(v) \mid v \text{ can reach } u \text{ in } G\}$. Describe a linear time algorithm that given G and b values in an array B outputs $a(u)$ for each $u \in V$.

   Solution: Construct the meta graph $G^{\text{SCC}} = (V', E')$ of strongly connected components, and for each component $C \in V'$, compute $\beta(C) := \min_{u \in C} \{b(u)\}$, the smallest label that appears in the component. Also for every component C, let $\alpha(C) = \min\left\{\beta(A) \mid A \text{ can reach } C \text{ in } G^{\text{SCC}}\right\}$. Notice that for any vertex $u \in V$, $a(u) = \alpha(C_u)$, where C_u is u's strongly connected component. Now that we have reduced to a DAG, we may use dynamic programming: our subproblem is simply computing α, and it obeys the recurrence

   $$\alpha(C) = \min \left\{ \overbrace{\beta(C)}^{\text{paths from within the SCC}} \quad , \quad \overbrace{\min_{A \to C \in E'} \alpha(A)}^{\text{paths from different SCCs}} \right\}$$

   where we say $\min \varnothing = \infty$. Subproblems depend on antineighbors of nodes, so a topological sorting of G^{SCC} gives a valid evaluation order (all edges go forward). We finally iterate through all $u \in V$ and set $a(u) = \alpha(C_u)$.

   Runtime: Constructing G^{SCC} and computing $\beta(C)$ for each $C \in V'$ both take linear time. Each edge in E' is considered in exactly one subproblem when taking the minimum, and there are $|V'|$ subproblems, so the dynamic programming algorithm also takes $O(|V'| + |E'|) = O(|V| + |E|)$ time. ■

   Rubric: 1 point for correctly applying SCC transform, 2 points for the algorithm. $-0.5(-1)$ pts for every minor(major) mistake.

   (b) (3 pts) Let $G = (V, E)$ be a directed graph. We have seen a linear-time algorithm that checks whether G is strongly connected. Suppose G is not strongly connected. Describe a linear time algorithm to check if one can add at most *one* edge to G to make it strongly connected.

   Solution: The problem is identical to checking if G^{SCC} can be made strongly connected by adding one edge, which is true if and only if G^{SCC} has exactly one source and sink. This may be checked in linear time by constructing G^{SCC} and iterating through vertices, counting those with in/out degree 0.

   For completeness, we'll prove the more general fact:

   Claim 1. *The minimum number of edges needed to make a (nontrivial) DAG with i sources and j sinks strongly connected is* $\max\{i, j\}$. [a]

   Proof: We'll prove the bounds separately.

Lower bound: For the graph to be strongly connected, every vertex must have at least one incoming and one outgoing edge. In particular, each source must have an added incoming edge and each sink must have an added outgoing edge. Thus, we need at least $\max\{i, j\}$ added edges; otherwise there'd be some old source(sink) without an incoming(outgoing) edge.

Upper bound: Ensuring a graph is strongly connected is equivalent to ensuring that every sink in the original graph can reach every source in the original graph. To see this, consider two vertices $u, v \in V$. There is some (old) source x that reaches u and some (old) sink y that reaches v, so if y can reach x, we have a path from u to v: $u \rightsquigarrow y \rightsquigarrow x \rightsquigarrow v$.

Let $G = (V, E)$ be a DAG with i sources and j sinks. We induct on $k = \max\{i, j\}$. Suppose we may strongly connect any DAG with $i' < k$ sources and $j' < k$ sinks with $\max\{i', j'\}$ new edges.

First suppose $i = j = k$. There are two possibilities:

- Every sink is reachable by every source (note that this covers the base case of $i = j = 1$). Denote the sources by $\{x_1, \ldots x_k\}$ and the sinks by $\{y_1, \ldots y_k\}$. Then, for every $1 \leq \ell \leq k-1$, add the edge $y_\ell \to x_{\ell+1}$. Also add the edge $y_k \to x_1$. We then have a cycle

  $$x_1 \rightsquigarrow y_1 \to x_2 \rightsquigarrow \cdots \rightsquigarrow y_{k-1} \to x_k \rightsquigarrow y_k \quad (\to x_1)$$

  So we have made every source reachable from every sink by adding k edges.
- There is a source x and a sink y such that x cannot reach y. Add the edge $y \to x$ to form G'.
  We argue that G' is a DAG. Consider any two vertices $u, v \in V$ where u can reach v. Suppose v can now reach u after adding $y \to x$. Since G was previously a DAG, the path must use $y \to x$, so we have a path $v \rightsquigarrow y \to x \rightsquigarrow u$. However, since we assumed u can reach v, we have a path $x \rightsquigarrow u \rightsquigarrow v \rightsquigarrow y$ which does not utilize the new edge. This is a contradiction, since x cannot reach y.
  Thus, G' is a DAG with all the same sources and sinks as G, except for x and y, so it has $k-1$ sources and $k-1$ sinks, which by the inductive hypothesis, can be strongly connected by adding $k-1$ more edges.

Next, suppose $i \neq j$. Without loss of generality, assume there are more sinks than sources (if not, reverse the graph): $i < j = k$. Then, pick any two sinks x and y and add the edge $x \to y$ to form G'. Now G' is a DAG (no cycles can include sinks) with one less sink than G. So, by the inductive hypothesis, we may strongly connect G' by adding $\max\{i, j-1\} = k-1$ more edges. □

■

[a] A formal proof of this fact is not necessary for credit.

Rubric: 3 points for single source/sink check and reasonable justification. $-0.5(-1)$ pts for every minor(major) mistake.

(c) (4 pts) Let $G = (V, E)$ be a directed graph in which each edge $e \in E$ has a non-negative reward $p(e)$ that can be collected by traversing it. Describe a linear time algorithm that given G, the edge rewards, and a starting node s, computes the maximum reward that a walk starting at s can collect. Note the the reward on an edge e is counted only the first time it is traversed in the walk since it is gone after it is picked up.

Solution (Direct DP): Notice that once we visit a vertex, we may traverse every edge in the vertex's strongly connected component without changing which vertices are reachable. So the optimal walk will be some path through G^{SCC} that collects the reward for every edge that in each strong connected component it visits. To that end, we construct $G^{\text{SCC}} = (V', E')$ and assign the following edge and vertex rewards:

- For every $C \in V'$, we compute $P(C) = \sum_{\substack{u \to v \in E \\ u,v \in C}} p(u \to v)$, the total reward from traversing all edges in C.
- For every $C \to D \in E'$, set $p(C \to D) = \max_{\substack{u \to v \in E \\ u \in C, v \in D}} p(u \to v)$, the highest reward that can be collected when traveling from C to D.

Now define *MaxReward*(C) to be the maximum reward that a path starting at $C \in V'$ can collect, where the reward of a path $C_1, \ldots, C_k$ in G^{SCC} is

$$\sum_{i=1}^{k} P(C_i) + \sum_{i=1}^{k-1} p(C_i \to C_{i+1})$$

Letting S be the component containing s, we want to evaluate *MaxReward*(S). The function obeys the recurrence

$$\textit{MaxReward}(C) = P(C) + \max_{C \to D \in E'} \{p(C \to D) + \textit{MaxReward}(D)\}$$

where we say $\max \varnothing = 0$ so that the definition is valid for sinks. Subproblems depend on a vertex's neighbors, so a reverse topological order on V' gives a valid evaluation order.

Runtime: Once again, computing G^{SCC} and the edge/metavertex rewards takes linear time.[a] Each edge is considered in exactly one subproblem, so the dynamic programming algorithm also runs in linear time. ■

[a] Begin with $P(C) = 0$ and $p(e) = 0$ for all $C \in V', e \in E'$. Label all the vertices by the index of their component in linear time. Then for each edge $u \to v \in E$, let $C_u, C_v \in V'$ be the corresponding components. If $C_u = C_v$, add $p(u \to v)$ to the component's reward $P(C_u)$. Otherwise, set $p(C_u \to C_v)$ to $\max\{p(C_u \to C_v), p(u \to v)\}$.

Solution (Longest Path): We may alternatively reduce to a standard longest path computation on a DAG (which we have seen is solvable in linear time). Compute G^{SCC} and its labels as above. We show two graphs, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, which are constructible in linear time and on which the problem may be solved.

In G_1, we set edge weights to account for both the reward on the edge and the reward on the component: $V_1 = V'$, $E_1 = E'$, and $w(C \to D) = p(C \to D) + P(D)$. Then compute the longest path length starting from S–call it ℓ. We need to additionally account for the reward of the first component, so we return $\ell + P(S)$.

In G_2, we bifurcate vertices to capture component rewards: $V_2 = V' \times \{0, 1\}$. For every edge $C \to D \in E'$, add the edge $(C, 1) \to (D, 0)$ to E_2 with weight $p(C \to D)$, and for every component $C \in V'$, add the edge $(C, 0) \to (C, 1)$ with weight $P(C)$. The maximum reward path from s in G is then given by the longest path in G_2 starting from $(S, 0)$. ■

Rubric: 2 points for correctly applying SCC transform, 2 points for the algorithm. $-0.5(-1)$ pts for every minor(major) mistake.

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.

   (a) (2.5 pts) For a parameter $k \geq 1$ the k-norm length of a path $v_1, v_2, \ldots, v_h$ 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 an integer $k \geq 1$ 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 $d_k(v_1, v_h)$ 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, $d_k(u, v) \leq d_k(x, y)$ if and only if $d_k(u, v)^k \leq d_k(x, y)^k$. Furthermore, we observe directly that $d_k(v_1, v_h)^k = \sum_{i=1}^{h-1} \ell(v_i \to v_{i+1})^k$.

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* $d_k(u, v)^\ell$. 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 $d_k(s, t)^k$ instead of $d_k(s, t)$. 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.

■

Rubric: 2 points for correct algorithm/analysis. 0.5 pts for exmaple.

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 $d_k(s,v)$ were the *true* k-norm distance from s to v, the analysis from the previous solution states that $d_k(s,v)^k = d_k(s,u)^k + \ell(u{\to}v)^k$. In summary, if we maintain a guess $kdist^k(s,v)$ of $d_k(s,v)^k$, 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 $d_k(s,v)^k$ instead of $d_k(s,v)$. Trying to maintain a guess for $d_k(s,v)$ 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 ← ∅
  while Q not empty
      (u, kdist^k(s,u)) ← ExtractMin(Q)
      X ← X ∪ {u}
      for each edge u→v where v ∉ X
          DecreaseKey(Q, min{GetVal(Q, v), kdist^k(s,u) + ℓ(v_h→v_{h+1})^k})
```

■

(b) (2.5 pts) 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 $\boldsymbol{bottle(s, v) \leftarrow \min\{bottle(s, v), \max\{bottle(s, u), \ell(u \to v)\}\}}$.

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 ← ∅
  while Q not empty
      (u, bottle(s, u)) ← ExtractMin(Q)
      X ← X ∪ {u}
      for each edge u→v where v ∉ X
          DecreaseKey(Q, min{GetVal(Q, v), max{bottle(s, u), ℓ(u→v)}})
```

■

(c) (2.5 pts) We will consider an alternate algorithm to compute the s-t bottleneck shortest path distance. Given G, s,t and a value $\lambda \geq 0$, describe a reduction to s-t reachbility to decide whether there is a path from s-t with bottleneck length at most λ. Use this and binary search to find the s-t bottleneck distance.

Solution: We utilize the natural graph transformation: Define $G' = (V, E')$ by

$$E' = \{u \to v \in E \mid \ell(u \to v) \leq \lambda\}$$

Then checking if we have an s-t path with bottleneck length at most λ is equivalent to checking if s can reach t in G', which we can do with WFS in linear time.

Binary Search: As usual, let $n = |V|, m = |E|$. Let *PathExists*(G, s, t, λ) denote the subroutine descibed above. First, re-label the weights by their ranks in the set of all edge weights: the smallest gets 1, the second smallest gets 2, and so on up to m. This can be done in $O(m \log m)$ time by sorting the list of weights. If there were duplicate weights, we'd have to be more careful. We now binary search on the weights to see the smallest bottleneck that admits a path from s to t:

```
Bottleneck(G, s, t):
  W[1..m] ← weights in G sorted in ascending order
  Re-label weights by their index in W
  if not PathExists(G, s, t, m) return ∞
  low ← 1, high ← m
  while low < high:
      mid ← ⌊(low+high)/2⌋
      if PathExists(G, s, t, mid)
          high ← mid
      else
          low ← mid + 1
  return W[low] ⟨⟨Translate back to original weights⟩⟩
```

Runtime: The running time of our algorithm is

$$O\left(\overbrace{m \log m}^{\text{Re-labeling}} + \overbrace{(n+m)}^{\textit{PathExists}} \overbrace{\log m}^{\text{Iterations}}\right) = O((n+m)\log m)$$

Which simplifies to $\boldsymbol{O((n+m)\log n)}$, since $\log m \leq \log n^2 = O(\log n)$.

Why not directly binary search on the weights? Directly binary searching on the weights may not work if (i) the weights are not integers or (ii) the maximum weight is not polynomial in n.

i. If the weights are not integers, we would have to continue the binary search until exactly one weight w lies in $[\textit{low}, \textit{high}]$. To see the problem with this, consider a graph whose edge weights are $\{1, 2, 3, 3 + 2^{-\text{bajillion}}, 4, 5\}$. Suppose the bottleneck distance between s and t was 3. Then, to have exactly one of 3 and $3 + 2^{-\text{bajillion}}$ in $[\textit{low}, \textit{high}]$, we'd have to perform $\log(4/2^{-\text{bajillion}}) \approx$ a bajillion iterations!

ii. Similarly, suppose the maximum weight was 2^n. Then, in the worst case, we'd have to perform $O(\log(2^n)) = O(n)$ iterations of binary search, leaving us with a runtime of $O(n^2 + nm)$.

The only[a] case in which directly binary searching on the weights is superior is when the weights are integers and are bounded by something sublinear in n like $\log n$. In this specific case, we'd achieve a running time of $O((n+m)\log\log n)$. In this problem, this is impossible, since we assume all edge weights are distinct. ■

[a] There are similar examples that use noninteger weights. Try to make one!

(d) (2.5 pts) Now consider another motivaton. Suppose each edge $e \in E$ has a probability $p(e)$ of failing. Given a path $v_1, v_2, \ldots, v_h$, what is the probability that none of the edges in the path fail assuming that the edges fail independently? Describe an algorithm to find the s-t path with the least probability of failing.

Solution (Reweighting the graph): For a path $v_1, \ldots v_h$ to succeed, all the edges should not fail, so we want to find the path that maximizes

$$(1 - p(v_1 \to v_2))(1 - p(v_2 \to v_3)) \cdots (1 - p(v_{h-1} \to v_h)) = \prod_{i=1}^{h-1} (1 - p(v_i \to v_{i+1}))$$

We then see that

$$\begin{aligned}
&\prod_{i=1}^{h-1} (1 - p(v_i \to v_{i+1})) \text{ is maximized} \\
\Leftrightarrow &\log \left(\prod_{i=1}^{h-1} (1 - p(v_i \to v_{i+1})) \right) = \sum_{i=1}^{h-1} \log(1 - p(v_i \to v_{i+1})) \text{ is maximized} \\
\Leftrightarrow &\sum_{i=1}^{h-1} [-\log(1 - p(v_i \to v_{i+1}))] \text{ is minimized}
\end{aligned}$$

So the problem is equivalent to the standard shortest path problem with edge weights of $\ell(e) = -\log(1 - p(e))$. Notice that the derived weights are positive:

$$0 \leq p(e) \leq 1 \Rightarrow 0 \leq 1 - p(e) \leq 1 \Rightarrow \log(1 - p(e)) \leq 0 \Rightarrow -\log(1 - p(e)) \geq 0$$

So we may simply run Dijkstra's algorithm on the modified graph to get a running time of $O(|E| \log |V|)$. ■

Solution (Modifying Dijkstra's Algorithm): For the numerical purists, we may directly modify Dijkstra's algorithm to remove the reliance on taking logs. For every vertex, we maintain a guess $pfail(s, v)$ of the probability of the most reliable path from s to v failing. We relax an edge $u \to v$ via

$$\boldsymbol{pfail(s, v) \leftarrow \min \{pfail(s, v), 1 - (1 - pfail(s, u))(1 - p(u \to v))\}}$$

Again, 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):

```
ProbabilisticDijkstra(G, s):
  Q ← MakePQ()
  Insert(Q, (s, 0))
  for each vertex v ≠ s
      Insert(Q, (v, 1))
  X ← ∅
  while Q not empty
      (u, pfail(s, u)) ← ExtractMin(Q)
      X ← X ∪ {u}
      for each edge u→v where v ∉ X
          DecreaseKey(Q, min{pfail(s, v), 1 − (1 − pfail(s, u))(1 − p(u→v))})
```

■

Rubric: For b-d: scaled graph reduction rubric if appropriate; otherwise 2 points for algorithm, 0.5 points for time analysis, $-0.5(-1)$ pts for each minor(major) error.

Standard graph-reduction rubric. 10 points =

- + 3 for constructing the correct graph.
  - +1 for correct vertices
  - +1 for correct edges
  - −½ for forgetting "directed" if the graph is directed
  - +1 for correct weights, costs, labels, or other annotations, if any
  - ○ The vertices, edges, and so on must be described as explicit functions of the input data.
  - ○ For most problems, the 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.
- + 3 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 the start vertex $(0, 0, 0)$ any target vertex of the form $(k, \cdot, \cdot)$ or $(\cdot, k, \cdot)$ or $(\cdot, \cdot, k)$." or "There is a French-flag walk from s to t in G if and only if $(s, 0)$ can reach $(t, 0)$ in H."
  - – No points for just writing (for example) "shortest path" or "reachability". 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" is not a problem!
- + 2 for correctly applying the correct **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 "Whatever-first search in H.")
  - −1 for using a slower algorithm than necessary, for example, Dijkstra's algorithm instead of breadth-first search.
  - −1 for explaining an algorithm from lecture or the textbook instead of just invoking it as a black box.

+ 2 for time analysis 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. Really, just do it the other way.

Standard dynamic programming rubric. 10 points =

- 3 points for a clear and correct English description of the recursive function you are trying to evaluate. (Otherwise, we don't even know what you're *trying* to do.)
  - − 1 for naming the function "OPT" or "DP" or any single letter.
  - – No credit if the description is inconsistent with the recurrence.
  - – No credit if the description does not explicitly describe how the function value depends on the named input parameters.
  - – No credit if the description refers to internal states of the eventual dynamic programming algorithm, like "the current index" or "the best score so far". The function must have a well-defined value that depends *only* on its input parameters (and constant global variables).
  - – An English explanation of the *recurrence* or *algorithm* does not qualify. We want a description of *what* your function returns; points for an explanation of *how* that value is computed are assigned in other items.
- 4 points for a correct recurrence, described either using mathematical notation or as pseudocode for a recursive algorithm.
  - + 1 for base case(s). −½ for one *minor* bug, like a typo or an off-by-one error.
  - + 3 for recursive case(s). −1 for each *minor* bug, like a typo or an off-by-one error.
  - − 2 for greedy optimizations without proof, even if they are correct.
  - – **No credit for iterative details if the recursive case(s) are incorrect.**
- 3 points for iterative details
  - + 1 for describing an appropriate memoization data structure. **Hash tables are NOT an appropriate memoization data structure!**
  - + 1 for describing a correct evaluation order; a clear picture is usually sufficient. If you use nested for loops, be sure to specify the nesting order. (In particular, if you draw a rectangle for a 2d array, be sure to label and direct the row and column indices.)
  - + 1 for correct time analysis. (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 specifically says otherwise.
- Iterative pseudocode is ***not*** required for full credit, provided the other details of your solution are clear and correct. We usually give two official solutions, one with pseudocode and one without.

If your solution does includes iterative pseudocode, you do not need to separately describe the recurrence, memoization structure, or evaluation order. But you ***do*** still need and English description of the underlying recursive function (or equivalently, the contents of the memoization structure). ***Perfectly correct iterative pseudocode, with no explanation or time analysis, is worth at most 6 points out of 10***, or at most 5 points out of 10 if you name the memoization array "DP".

- Partial credit for incomplete solutions depends on the running time of the ***best possible*** completion (up to the target running time). For example, consider a solution that contains *only* a clear English description of a function, with no recurrence or iterative details. If the described function *can* be developed into an algorithm with the target running time, the solution is worth 3 points; however, if the function leads to an algorithm that is slower than the target time by a factor of n, the solution could be worth only 2 points ($=$ 70% of 3, rounded).