1) 24.1-3
2) 24.1-4
3) 24.2-4
4) 24.3-2
5) 24.3-10
6) 24-2
7) 25.1-9
8) 25.2-6
9) 25.2-7
10) 25-1
(1) CLR 24.1-3

The proof of Lemma 24.2 shows that for every $v$, $d[v]$ has attained its final value after length (any shortest-weight path to $v$) iterations of BELLMAN-FORD. Thus after $m$ passes, BELLMAN-FORD can terminate. We don’t know $m$ in advance, so we can’t make the algorithm loop exactly $m$ times and then terminate. But if we just make the algorithm stop when nothing changes any more, it will stop after $m+1$ iterations (i.e., after one iteration without a change), unless there is a negative weight cycle. Hence, we should also make sure that the number of iterations does not exceed $|V[G]| - 1$.

BELLMAN-FORD-$(M + 1)(G, w, s)$
1. INITIALIZE-SINGLE-SOURCE($G, s$)
2. changes $\leftarrow$ TRUE
3. iteration $\leftarrow$ 1
4. while (changes = TRUE) $\land$ (iteration $\leq$ $|V[G]| - 1$)
   do changes $\leftarrow$ FALSE
   iteration $\leftarrow$ iteration + 1
7. for each edge $(u, v) \in E[G]$
   do RELAX-M($u, v, w$)
9. for each edge $(u, v) \in E[G]$
10. do if $d[v] > d[u] + w(u, v)$
11. then return FALSE
12. return TRUE

RELAX-M($u, v, w$)
1. if $d[v] > d[u] + w(u, v)$
2. then $d[v] \leftarrow d[u] + w(u, v)$
3. $\pi[v] \leftarrow u$
4. changes $\leftarrow$ TRUE

(2) CLR 24.1-4

Change line 7 of the BELLMAN-FORD algorithm, as given in the CLR text, to:

then $d[v] \leftarrow -\infty$

(3) CLR 24.2-4

Consider a node $v$ in a directed acyclic graph $G$. The paths in $G$ starting from $v$ must go from any outgoing edge from $v$. Let $< V, u >$ be one of such edges, then the paths starting from $v$ and containing $< v, u >$ can either stop at $u$ or continue from $u$
to some other vertices. Since \( G \) is a directed acyclic graph, any edge can only appear at most once in any path. We can thus conclude that the number of paths starting from \( v \) and containing \( \langle v, u \rangle \) equals the number of paths starting from \( u \) plus one. We give the algorithm for calculating the total number of paths in \( G \) below.

\[ \text{DAG-COUNTPATHS-(G)} \]

1. topologically sort the vertices of \( G \)
2. \( \text{totalcount} \leftarrow 0 \)
3. for every vertex \( v \) in \( G \), taken in reverse topologically sorted order
   4. \( \text{do} \ \text{count}[v] \leftarrow 0 \)
   5. \( \text{for} \) each \( u \in \text{Adj}[v] \)
   6. \( \text{do} \ \text{count}[v] \leftarrow \text{count}[v] + \text{count}[u] + 1 \)
   7. \( \text{totalcount} \leftarrow \text{totalcount} + \text{count}[v] \)

The running time of this algorithm is \( \Theta(V + E) \).

(4) CLR 24.3-2

Consider the graph below:

![Graph diagram](attachment:graph.png)

The predecessor subgraph returned by Dijkstra's algorithm will look like this:
The correct answer should use the path \( s \rightarrow x \rightarrow t \rightarrow u \) instead of \( s \rightarrow u \), since \( \delta(s, u) = 0 \).

In Theorem 24.6, the assumption of no negative weights is used to conclude that \( \delta(s, y) \leq \delta(s, u) \) and \( d[y] \leq d[u] \). If there are negative weights in the path \( p \), \( \delta(s, y) \) may be greater than \( \delta(s, u) \), and the contradiction will not hold, as \( d[u] \) can be greater than \( \delta(s, u) \) and hence \( d[u] \neq \delta(s, u) \).

(5) CLR 24.3-8

We use the loop invariant:

At the start of each iteration of the while loop of lines 4-8, \( d[v] = \delta(s, v) \) for each vertex \( v \in S \).

**Initialization:** Initially \( S = \phi \) so the invariant is trivially true.

**Maintenance:** To prove by contradiction, assume that \( u \) is the first vertex added to \( S \) such that \( d[u] \neq \delta(s, u) \). We know that \( u \neq s \) because \( s \) is the first vertex added to \( S \) and \( d[s] = \delta(s, s) = 0 \) at that time. As \( u \neq s \) we know that \( S \neq \phi \) when \( u \) is added.

There is a shortest-path \( p \) from \( s \) to \( u \) in \( G \), otherwise \( d[u] = \delta(s, u) = \infty \) from the very beginning, contradicting our initial assumption. Say on path \( p \), when going from \( s \) to \( u \), \( y \) is the first vertex in \( V - S \). Also, \( x \) is the predecessor of \( y \) along this path.

\( d[y] = \delta(s, y) \) when \( u \) is added to \( S \). To see why this is so we observe that \( x \) was added to \( S \) before \( u \). And when \( x \) was added to \( S \) the edge \( (x, y) \) was relaxed and since \( d[x] = \delta(s, x) \) at that moment, therefore \( d[y] \) gets set to \( \delta(s, y) \).

No edge along the path from \( y \) to \( u \) has a negative weight because all negative edges must be of the form \((s, -)\) and \( s \neq y \). Because \( y \) occurs before \( u \) on a path with no negative-weight edges \( \delta(s, y) \leq \delta(s, u) \). Then we have

\[ d[y] = \delta(s, y) \leq \delta(s, u) \leq d[u] \]
Now both $y$ and $u$ were in $V - S$ when $u$ was chosen as the next vertex to add to $S$ therefore

$$d[u] \leq d[y]$$

From the above two equations we have $d[u] = \delta(s, u)$ contradicting our intial assumption.

**Termination:** $Q = \emptyset$ and $S = V$, thus $d[v] = \delta(s, v)$ for all $v \in S$ and the shortest path has been solved.

(6) CLR 24-2

(a) Say that $x$ nests in $y$ and $y$ nests in $z$. This means that we can formulate a $\pi$ mapping for both nesting relaitons such that:

$$x_{\pi_i} < y_i \text{ and } y_{\pi_j} < z_j$$

Because all values of $i$ and $j$ are unique and drawn from the set $1...d$ we can find for each $i$ a $j$ such that $i = \pi_j$. Then we have:

$$x_{\pi_i} < y_i = y_{\pi_j} < z_j$$

$$x_{\pi_i} < z_j$$

And we can find such a unique $z_j$ for every $x_{\pi_i}$. Thus the relation is transitive.

(b) Sort in non-decreasing order the $d$ dimension values within both $x = (x_1, x_2...x_d)$ and $y = (y_1, y_2...y_d)$. Compare $x_i$ and $y_i$ for every value of $i = 1$ to $d$. If $x_i < y_i$ for all $i$ then $x$ nests within $y$.

(c) Sort the $d$ dimension values within each box $B_i$. This takes $O(d \log d)$ for every box, totalling to $O(nd \log d)$ for the $n$ boxes. For each possible pair of box $(B_i, B_j)$ check if $B_i$ nests in $B_j$ or if $B_j$ nests in $B_i$. Each pair-comparison takes $O(d)$ and there are $O(n^2)$ pairs in total, so all nesting pair relations can be obtained in $O(dn^2)$ steps. We create a graph with $n$ nodes each representing one of the $B_i$'s. For each $(B_i, B_j)$ pair such that $B_i$ nests in $B_j$ we add a directed edge from $B_i$ to $B_j$. This graph construction takes $O(n^2)$ time. Now add two new vertices $s$ and $d$ to the graph, such that $s$ has an outgoing edge to each vertex and $d$ had an incoming edge from each vertex.

Run topological sort on the resulting graph with $s$ as the root vertex. Since $|V| = O(n)$ for this graph, the running time for topological sort is $O(n + n^2)$. Now consider all the vertices in topological order. For each vertex $u$ examine its adjacency list and for each edge $(u, v)$ “anti-relax” the edge as:

```
if d[v] < d[u] + w(u, v)
then d[v] = d[u] + w(u, v)
```
\[ \pi[v] = u \]

This traversal takes \( O(E) = O(n^2) \) time. When the traversal terminates, find the vertex \( v \) with the largest \( d[v] \) value in \( O(n) \) time. Remove \( s \) from the graph and call PRINT-PATH on the vertex \( v \).

the overall running time of the algorithm is bounded by \( O(dn^2) \)

(7) CLR 25.1-9

The presence of a negative-weight cycle can be determined by looking at the diagonal of the matrix \( L^{(n-1)} \) computed by an all-pairs shortest-path algorithm. If the diagonal contains any negative number there must be a negative-weight cycle.

(8) CLR 25.2-6

There are several ways to use FLOYD-WARSHALL to detect negative-weight cycles:

- If we modify the algorithm to calculate \( \Pi^{[k]} \), then while FLOYD-WARSHALL is running, if any element \( \Pi[i,i] \) is not NIL, then there exists some node that is the predecessor of the node \( i \) in a path from \( i \) to \( i \). Since the distance from a node to itself is 0, there must be some other path from the node to itself. Since this path must have a weight less than 0, it is a negative weight cycle.

- Similarly, if any element on the diagonal of the \( D \) matrix is non-zero, then there must be a negative weight cycle including that vertex, for reasons as above.

- The algorithm could be altered to perform one more iteration (e.g., by invoking EXTEND-SHORTEST-PATHS(\( D^{[n]} \), \( W \)) to detect any changes by comparing \( D^{[n]} \) and \( D^{(n+1)} \). If any distances are changed, then there must be a negative weight cycle.

(9) CLR 25.2-7

Initialize the \( \phi \) matrix as follows:
\[ \phi_{ij}^{(0)} = \text{NIL if there is no edge } (i, j) \]
\[ \phi_{ij}^{(0)} = \text{DIRECT if there is an edge } (i, j) \]

We can recursively define \( \phi_{ij}^{(k)} \) as follows:
\[ \phi_{ij}^{(k)} \leftarrow k \text{ if } d_{ik}^{(k-1)} + d_{kj}^{(k-1)} < d_{ij}^{(k-1)} \]
\[ \phi_{ij}^{(k)} \leftarrow \phi_{ij}^{(k-1)} \text{ otherwise} \]

For modifying the Floyd-Warshall algorithm, insert the following between line 6 and 7:
if \(d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\)
\[\phi_{ij}^{(k)} \leftarrow k\]
else
\[\phi_{ij}^{(k)} \leftarrow \phi_{ij}^{(k-1)}\]

And the modified version of printing a shortest path from the all-pairs algorithm:

PRINT-ALL-PAIRS-PATH-2(\(\phi, i, j\))
1. if \(\phi_{ij}^{(n)} = \text{DIRECT E}\)
2. then PUSH(S, i)
3. PUSH(S, j)
4. else if \(\phi_{ij}^{(n)} = \text{NIL}\)
5. then print “No path from \(i\) to \(j\) exists”
6. else PRINT-ALL-PAIRS-PATH-2(\(\phi^{(n)}, i, \phi_{ij}^{(n)}\))
7. POP(S)
8. PRINT-ALL-PAIRS-PATH-2(\(\phi^{(n)}, \phi_{ij}^{(n)}, j\))
9. while !empty(S)
10. \(x \leftarrow \text{POP}(S)\)
11. print \(x\)

(10) CLR 25-1

(a) Let \(T\) be the \(|V| \times |V|\) matrix representing the transitive closure, such that \(T[i, j] = 1\) if there is a path from \(i\) to \(j\), and 0 if not. Initialize \(T\) (when there are no edges in \(G\)) as follows:
\[
T[i, j] = \begin{cases} 
1 & \text{if } i = j \\
0 & \text{otherwise}
\end{cases}
\]

\(T\) can be updated as follows when an edge \((u, v)\) is added to \(G\):

TRANSITIVE-CLOSURE-UPDATE(u, v)
1. for \(i \leftarrow 1\) to \(|V|\)
2. do for \(j \leftarrow 1\) to \(|V|\)
3. do if \(T[i, u] = 1\) and \(T[v, j] = 1\)
4. then \(T[i, j] \leftarrow 1\)

1. This says that the effect of adding edge \((u, v)\) is to create a path (via the new edge) from every vertex that could already reach \(u\) to every vertex that could already be reached from \(v\).
2. Note that the procedure sets \(T[u, v] \leftarrow 1\), since the initial values \(T[u, u] = T[v, v] = 1\).
3. This takes \( \Theta(V^2) \) time because of the two nested loops.

(b) Consider inserting the edge \( v_n \rightarrow v_1 \) into the straight-line graph \( v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_n \), where \( n = |V| \). Before this edge is inserted, only \( \frac{n(n+1)}{2} \) entries in \( T \) are 1 (the entries on and above the main diagonal). After the edge is inserted, the graph is a cycle in which every vertex can reach every other vertex, so all \( n^2 \) entries in \( T \) are 1. Hence \( n^2 - \frac{n(n+1)}{2} = \Theta(n^2) = \Theta(V^2) \) entries must be changed in \( T \), so any algorithm to update the transitive closure must take \( \Omega(V^2) \) time on this graph.

(c) The algorithm in part (a) would take \( \Theta(V^4) \) time to insert all possible \( \Theta(V^2) \) edges, so we need a more efficient algorithm in order for any sequence of insertions to take only \( O(V^3) \) total time.

To improve the algorithm, notice that the loop over \( j \) is pointless when \( T[i,v] = 1 \). That is, if there is already a path from \( i \) to \( v \), then adding the edge \( u \rightarrow v \) can’t make any new vertices reachable from \( i \). The loop to set \( T[i,j] \) to 1 for \( j \) such that there’s a path from \( v \) to \( j \) is just setting entries that are already 1. Eliminate this redundant processing as follows:

\[
\text{Transitive-Closure-Update}(u,v) \\
1. \text{for } i \leftarrow 1 \text{ to } |V| \\
2. \quad \text{do if } T[i,u] = 1 \text{ and } T[i,v] = 0 \\
3. \quad \quad \text{then for } j \leftarrow 1 \text{ to } |V| \\
4. \quad \quad \quad \text{do if } T[v,j] = 1 \\
5. \quad \quad \quad \quad \text{then } T[i,j] \leftarrow 1
\]

We show that this takes \( O(V^3) \) time to update the transitive closure for any sequence of \( n \) insertions:

1. There can’t be more than \( |V|^2 \) edges in \( G \), so \( n \leq |V|^2 \).
2. Summed over \( n \) insertions, the time to execute lines 1 and 2 is \( O(nV) = O(V^3) \).
3. Lines 3-5, which take \( \Theta(V) \) time, are executed only \( O(V^2) \) times for \( n \) insertions. To see this, notice that lines 3-5 are executed only when \( T[i,v] = 0 \), and in that case line 5 sets \( T[i,v] \leftarrow 1 \), so the number of 0 entries in \( T \) is reduced by at least one each time lines 3-5 run. Since there are only \( |V|^2 \) entries in \( T \), lines 3-5 can run at most \( |V|^2 \) times.
4. Hence the total running time over \( n \) insertions is \( O(V^3) \).