Exercises (not part of formal assignment, not graded)

1. Design an efficient algorithm that, when given a graph $G = (V, E)$ as input, determines whether $G$ contains a cycle.
   
   (a) Do this for the case where $G$ is an undirected graph.
   
   (b) Do this for the case where $G$ is a directed graph.

Solution:

(a) For an undirected graph, there exists a cycle if and only if there exists a back edge. An edge $(u, v)$ is a back edge if $\text{pre}(v) < \text{pre}(u) < \text{post}(u) < \text{post}(v)$. We can execute depth-first search (DFS) on the graph $G$ to check whether a back edge exists. If we come across a vertex $u$ where an adjacent vertex $v$ has already been visited and is not the parent of $u$, then the graph contains a cycle. Runtime complexity: $\mathcal{O}(|V| + |E|)$

(b) For a directed graph, there exists a cycle if there is a back edge that points to a visited vertex which is currently in the recursion stack of the DFS traversal. We need to keep track of the visited vertices currently in the recursion stack of function for DFS. While executing DFS if we visit a vertex which is in the current recursion stack, then there is a cycle in the given graph $G$. Runtime complexity: $\mathcal{O}(|V| + |E|)$

2. Execute Dijkstra’s algorithm on the weighted graph below, using the vertex $A$ as the source vertex. For each vertex in the graph, write down the shortest path cost from vertex $A$. Please understand how the vertex labels change as the algorithm is modified. Please also understand how vertices would be labeled with “predecessor in the shortest path found from the source” information that would enable reconstruction of source-sink shortest paths.
Solution:
The table below shows the shortest path cost and preceding vertex for each vertex.

<table>
<thead>
<tr>
<th>step</th>
<th>vertex</th>
<th>dist(A)</th>
<th>dist(B)</th>
<th>dist(C)</th>
<th>dist(D)</th>
<th>dist(E)</th>
<th>dist(F)</th>
<th>dist(G)</th>
<th>dist(H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>0</td>
<td>5/A</td>
<td>∞</td>
<td>∞</td>
<td>4/A</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>2</td>
<td>E</td>
<td>0</td>
<td>5/A</td>
<td>∞</td>
<td>∞</td>
<td>4/A</td>
<td>15/E</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
<td>0</td>
<td>5/A</td>
<td>11/B</td>
<td>∞</td>
<td>4/A</td>
<td>15/E</td>
<td>15/B</td>
<td>∞</td>
</tr>
<tr>
<td>4</td>
<td>C</td>
<td>0</td>
<td>5/A</td>
<td>11/B</td>
<td>12/C</td>
<td>4/A</td>
<td>15/E</td>
<td>15/B</td>
<td>28/C</td>
</tr>
<tr>
<td>5</td>
<td>D</td>
<td>0</td>
<td>5/A</td>
<td>11/B</td>
<td>12/C</td>
<td>4/A</td>
<td>15/E</td>
<td>15/B</td>
<td>28/C</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>0</td>
<td>5/A</td>
<td>11/B</td>
<td>12/C</td>
<td>4/A</td>
<td>15/E</td>
<td>15/B</td>
<td>28/C</td>
</tr>
<tr>
<td>7</td>
<td>G</td>
<td>0</td>
<td>5/A</td>
<td>11/B</td>
<td>12/C</td>
<td>4/A</td>
<td>15/E</td>
<td>15/B</td>
<td>28/C</td>
</tr>
<tr>
<td>8</td>
<td>H</td>
<td>0</td>
<td>5/A</td>
<td>11/B</td>
<td>12/C</td>
<td>4/A</td>
<td>15/E</td>
<td>15/B</td>
<td>28/C</td>
</tr>
</tbody>
</table>

3. Run the Bellman-Ford algorithm on the weighted graph below, using the vertex A as the source vertex. For each vertex in the graph, write down the shortest path cost from vertex A. Please understand how the algorithm “relaxes” constraints on number of edges in paths from A, i.e., effectively does a “successive approximation” of true shortest path costs from A.

Solution: The table below shows that shortest path cost from the vertex A for each vertex.

<table>
<thead>
<tr>
<th>k</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>

4. DPV 2.23 An array $A[1...n]$ is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and, if so, to find that element. The elements of the array are not necessarily from some ordered domain like the integers, and so there can be no comparisons of the form “is $A[i] > A[j]$?”. (Think of the array elements as GIF files, say.) However you can answer questions of the form: “is $A[i] = A[j]$?” in constant time.

(a) Show how to solve this problem in $O(n \log(n))$ time. (Hint: Split the array A into two arrays A1 and A2 of half the size. Does knowing the majority elements of A1 and A2 help you figure out the majority element of A? If so, you can use a divide-and-conquer approach.)

(b) Can you give a linear-time algorithm? (Hint: Here’s another divide-and-conquer approach:
- Pair up the elements of A arbitrarily, to get $n/2$ pairs
- Look at each pair: if the two elements are different, discard both of them; if they are the same, keep just one of them

Show that after this procedure there are at most $n/2$ elements left, and that they have a majority element if and only if A does.)
Solution:

(a) If there is a majority element in $A$, there must also be the majority element in either $A_1$ or $A_2$ or both.

- If there is the same majority element in both, we return it as the majority of $A$.
- if $A_1$ has a majority element $E_0$, we must check all elements in $A$ against $E_0$ to determine if $E_0$ is indeed a majority element of $A$. $O(n)$ runtime.
- if $A_2$ has a majority element $E_1$, we must check all elements in $A$ against $E_1$ to determine if $E_1$ is indeed a majority element of $A$. $O(n)$ runtime.

At each stage, the maximum occurs if both $A_1$ and $A_2$ have different majorities: $O(2n) = O(n)$

Pseudocode

```python
procedure majority(A[1..n])
    if n == 1:
        return A[1]
    Let A1 and A2 be the first and second halves of A
    M1 = majority(A1)
    M2 = majority(A2)
    if M1 is a majority of element A:
        return M1
    if M2 is a majority of element A:
        return M2
    return "no majority"
```

Running time: $T(n) = 2T(n/2) + O(n) = O(n \log n)$

(b) As described above, we pair up the elements of $A$ arbitrarily, to get $n/2$ pairs. For each pair, if the two elements are different, we discard both of them else we keep just one of them.

Pseudocode

```python
procedure majority(A[1..n])
    x = prune(A)
    if x is a majority element of A:
        return x
    else:
        return "no majority"

procedure prune(S[1..n])
    if n == 1:
        return S[1]
    if n is odd:
        if S[n] is a majority element of S:
            return S[n]
        n = n - 1
    S' = [ ] (empty list)
    for i = 1 to n/2:
        if S[2i - 1] = S[2i]:
            add S[2i] to S'
    return prune(S')
```
Justification: We’ll show that each iteration of the prune procedure maintains the following invariant: if $x$ is a majority element of $S$ then it is also a majority element of $S'$. The rest then follows. Suppose $x$ is a majority element of $S$. In an iteration of prune, we break $S$ into pairs. Suppose there are $k$ pairs of Type One and $l$ pairs of Type Two:

- Type One: the two elements are different. In this case, we discard both.
- Type Two: the elements are the same. In this case, we keep one of them.

Since $x$ constitutes at most half of the elements in the Type One pairs, $x$ must be a majority element in the Type Two pairs. At the end of the iteration, what remains are $l$ elements, one from each Type Two pair. Therefore $x$ is the majority of these elements.

Running time. In each iteration of prune, the number of elements in $S$ is reduced to $l \leq |S|/2$, and a linear amount of work is done. Therefore, the total time taken is $T(n) \leq T(n/2) + O(n) = O(n)$.

5. DPV 4.4: Here’s a proposal for how to find the length of the shortest cycle in an undirected graph with unit edge lengths. When a back edge, say $(v, w)$, is encountered during a depth-first search, it forms a cycle with the tree edges from $w$ to $v$. The length of the cycle is $\text{level}[v] - \text{level}[w] + 1$, where the level of a vertex is its distance in the DFS tree from the root vertex. This suggests the following algorithm:

- Do a depth-first search, keeping track of the level of each vertex.
- Each time a back edge is encountered, compute the cycle length and save it if it is smaller than the shortest one previously seen.

Show that this strategy does not always work by providing a counterexample as well as a brief (one or two sentence) explanation.

Solution: The graph in the figure below is a counterexample: vertices are labeled with their level in the DFS tree, back edges are dashed. The shortest cycle consists of vertices 1-4-5, but the cycle found by the algorithm is 1-2-3-4. In general, the strategy will fail if the shortest cycle contains more than one back edge.

Problems (must be written up and turned in)

1. DPV 4.3: Squares. Design and analyze an algorithm that takes as input an undirected graph $G = (V, E)$ and determines whether $G$ contains a simple cycle (that is, a cycle which doesn’t intersect itself) of length four. Its running time should be at most $O(|V|^3)$. You may assume that the input graph is represented either as an adjacency matrix or with adjacency lists, whichever makes your algorithm simpler.
Suppose the input graph \( G \) is given as an adjacency matrix. Notice that \( G \) contains a square if and only if there are two vertices \( u \) and \( v \) that share more than one neighbor. For any \( u, v \) we can check this in time \( O(|V|^2) \) by comparing the row of \( u \) and the row of \( v \) in the adjacency matrix of \( G \). Because we need to repeat this process \( O(|V|^2) \) to iterate over all \( u \) and \( v \), this algorithm has running time \( O(|V|^3) \). We can do better by noticing that, when comparing the rows of the adjacency matrix \( a_u \) and \( a_v \), we are actually checking if \( a_u \cdot a_v \) is greater than 1, i.e. if \( [A(G)^2]_{uv} \geq 1 \). It suffices then to compute \( A(G) \), which we can do in time \( O(|V|^2 \cdot 71) \) using our matrix multiplication algorithm and check all non-diagonal entries to see if we find one larger than 1.

2. Consider the modified binary search algorithm that splits the input not into two sets of almost-equal sizes, but into three sets of almost-equal sizes. Write down the recurrence relation for this ternary search algorithm and find the asymptotic complexity of this algorithm. Compare the number of comparisons made by this algorithm and binary search.

Solution: The recurrence for normal binary search is \( T_2(n) = T_2(n/2) + 1 \). This accounts for one comparison (on an element which partitions the \( n \)-element list of sorted keys into two \( n/2 \)-element sets) and then a recursive call on the appropriate partition. For ternary search we make two comparisons on elements which partition the list into three sections with roughly \( n/3 \) elements and recurse on the appropriate partition. Thus analogously, the recurrence for the number of comparisons made by this ternary search is: \( T_3(n) = T_3(n/3) + 1 \). However, just as for binary search the second case of the Master Theorem applies. We therefore conclude that

\[
T_3(n) \in \Theta(\log(n))
\]

Number of comparisons: In binary search we have \( \log_2(n) + O(1) \) number of comparisons since at each step we just need to compare the value of the middle element of the appropriate array. By similar reasoning, in ternary search we have \( 2 \log_3(n) + O(1) \) number of comparisons which is greater than the number of comparisons made in binary search.

3. Let \( v \) and \( w \) be two vertices in an unweighted, directed graph \( G = (V, E) \). Design an efficient linear-time algorithm to find the number of distinct shortest paths (not necessarily disjoint) from \( v \) to \( w \). For example, in the graph shown below, there are two distinct shortest paths from \( A \) to \( E \): \( A \rightarrow C \rightarrow D \rightarrow E \) and \( A \rightarrow B \rightarrow D \rightarrow E \). Justify why your algorithm works, give pseudocode, and give an analysis of runtime complexity.

Solution: Execute breadth-first search starting from the vertex \( v \). For each vertex, we need to keep track of the number of different shortest paths from the vertex \( v \) to it. Let \( \text{numpaths}[x] \) denote the number of shortest paths from \( v \) to \( x \) and let \( y \) be an adjacent vertex of \( x \) which is the current vertex. We have two cases:

- \( y \) has never been visited before. In this case, the number of shortest paths to \( y \) is equal to the number of shortest paths to \( x \)
• y has been visited before and a shortest path to it is possible through x. In this case, the number of shortest paths to y gets incremented by the number of shortest paths to x.

When w is encountered for the first time the level of w is the length of the shortest path and therefore, we can stop expanding vertices when we go past the length of the shortest path. We can finally return numpaths[w].

The correctness of the algorithm is embodied in its construction itself. In the first case, when y has not been visited before, all the shortest paths from v to y must go through the vertex x. Therefore, the number of shortest paths to y is equal to the number of shortest paths to x.

In the second case, when y has been visited before, we have found a new set of paths that can reach to y via x. Thus, the number of shortest paths to y gets incremented by the number of shortest paths to x.

The two cases correctly define the number of shortest paths for any neighbor y. Further, the vertices which are at a deeper level than that of w can never contribute to the number of different shortest paths to w, since we must check the shortest path length to w before incrementing.

Therefore, we can prune our search for such vertices. This completes the proof of correctness.

Pseudocode:

```
procedure distinct_paths(G)
Let dist[w] = shortest distance from v to w
dist[x]=∞ ∀x ∈ V \ w
dist[v]=0
Q.enqueue(v)
while Q is not empty
  x = Q.front()
  Q.dequeue()
  for all (x, y) ∈ E
    if dist[y] = ∞ and dist[x]+1 ≤ dist[w]
      dist[y] = dist[x]+1
      Q.enqueue(y)
      numpaths[y] = numpaths[x]
    else if dist[y] == dist[x]+1
      numpaths[y] = numpaths[y] + numpaths[x]
  return numpaths[w]
```

**Runtime Complexity** We can first find dist[w] by executing breadth-first search on the graph. Further, every vertex and every edge will be explored once in the worst case. Therefore, the runtime complexity of the algorithm is \(O(|V| + |E|)\).

4. Consider an edge-weighted, directed graph \(G = (V, E)\) in which the edge weights represent capacities of the edges. Given start and end vertices \(u, v ∈ V\), the maximum bottleneck path from \(u\) to \(v\) is the directed \(u\) to \(v\) path in \(G\) that maximizes the minimum weight of any edge in that path. Design an efficient algorithm that, for given \(G = (V, E)\) and \(u, v ∈ V\), finds a maximum bottleneck path from \(u\) to \(v\). Justify why your algorithm works, give pseudocode, and give an analysis of runtime complexity.

**Solution:** We can modify Dijkstra’s algorithm such that a vertex with a larger bottleneck path to it from \(u\) has a higher priority in the priority queue. Let pathto(x) denote the minimum weight
of any edge on the maximum bottleneck path from $u$ to $x$, found so far. Instead of updating the shortest distance for the neighboring vertices of the current vertex as we do in Dijkstra’s algorithm, we update their maximum bottleneck paths. For each neighbour $y$ of the current vertex $x$, we first find the new $\text{path}_t o[y]$ value considering the path that goes through $x$. If this value is greater than the existing one, we then update $\text{path}_t o[y]$ as shown in the pseudocode below.

The proof of the algorithm is very similar to the proof of Dijkstra’s algorithm discussed in class. Here the invariant is $\text{path}_t o[x]$ which is always an underestimate of the maximum bottleneck path from $u$ to $x$.

Pseudocode:

```
procedure maximum_bottleneck_path($G, u$)
    for all $x \in V$
        $\text{path}_t o[x] = 0$
        $\text{prev}[x] = \text{Undefined}$
        add $x$ to $PQ$
    $\text{path}_t o(u) = \infty$
    while $PQ$ is not empty
        $x = \text{vertex in } PQ \text{ with maximum } \text{path}_t o \text{ value}$
        remove $x$ from $PQ$
        for each edge $(x, y) \in E$
            if $\text{path}_t o[y] < \max(\text{path}_t o(x), \text{weight}(x, y))$
                $\text{path}_t o[y] = \max(\text{path}_t o(x), \text{weight}(x, y))$
                $\text{prev}[y] = x$
        return $\text{path}_t o[]$, $\text{prev}[]$
```

```
procedure path_from_u_to_v($\text{prev}[]$, $v$)
    Let $S$ be a stack
    while $\text{prev}[v]$ is defined
        $S$.push($v$)
        $v = \text{prev}[v]$
        $S$.push($v$)
    return $S$
```

**Runtime Complexity**: The runtime complexity of the algorithm is $O((|V|+|E|) \log(V))$ which is same as the runtime complexity of Dijkstra’s algorithm. The complexity does not change because there is just a modification in the values stored which does not have any effect on the runtime of the algorithm.

5. Let $G = (V, E)$ be an edge-weighted, directed graph. Given $G$ and an identified source vertex $s \in V$, we would like to find shortest paths from $s$ to all other vertices in $G$. However, if there exist multiple shortest paths from $s$ to any $v \in V$, we must identify a shortest path that has the least number of edges. Design an efficient algorithm that performs this task. Justify why your algorithm works, give pseudocode, and give an analysis of runtime complexity.

**Solution**: The problem can be solved by adding a simple modification to the Dijkstra’s algorithm. Let $\text{numedges}$ be an array such that $\text{numedges}(u)$ is the least number of edges on a shortest path from $s$ to $u$ known so far. Thus, $\text{numedges}$ is used as a tiebreaker when deciding the previous vertex on the shortest path to a given vertex.
In the update rule,

- if \( \text{dist}(v) + \text{weight}(v, w) = \text{dist}(w) \) and \( \text{numedges}(v) + 1 < \text{numedges}(w) \), then change \( \text{prev}(w) \) to \( v \) and \( \text{numedges}(w) \) to \( \text{numedges}(v) + 1 \).

- If \( \text{dist}(v) + \text{weight}(v, w) < \text{dist}(w) \), then update \( \text{prev}(w) \) and \( \text{dist}(w) \), and set \( \text{numedges}(w) \) to \( \text{numedges}(v) + 1 \).

The proof of correctness lies in the construction of the algorithm and the proof of Dijkstra’s algorithm discussed in class. If any neighbor \( w \) of current vertex \( v \) can be reached by more than one paths which have same distance, we take the one which has least number of edges. This is essentially done in the first update rule specified above. Second update rule updates the distance and previous values for the neighbor \( w \) if a shorter path is discovered. Further, in this case the number of edges on the shortest path to \( w \) is equal to the number of edges on the shortest path to \( v + 1 \) as the shortest path to \( w \) has to go through \( v \). The two conditions defined above correctly set the value of number of edges for the shortest path from \( s \) to any vertex \( w \). This completes the proof of correctness.

Pseudocode:

```plaintext
procedure path_with_minimum_edges(G, s)
    \( \text{dist}[s] = 0 \)
    \( \text{numedges}[s] = 0 \)
    add \((s, \text{dist}[s], \text{numedges}[s])\) to \( \text{PQ} \)
    for all \( x \in V \setminus s \)
        \( \text{dist}[x] = \infty \)
        \( \text{prev}[x] = \text{Undefined} \)
        \( \text{numedges}[x] = \infty \)
        add \((x, \text{dist}[x], \text{numedges}[x])\) to \( \text{PQ} \)
    while \( \text{PQ} \) is not empty
        \( x = \text{vertex in PQ with minimum dist value} \)
        remove \( x, \text{dist}[x], \text{numedges}[x] \) from \( \text{PQ} \)
        for each edge \((x, y)\) in \( E \)
            if \( \text{dist}[y] > \text{dist}[x] + \text{weight}(x, y) \)
                \( \text{dist}[y] = \text{dist}[x] + \text{weight}(x, y) \)
                \( \text{prev}[x] = y \)
                \( \text{numedges}[y] = \text{numedges}[x] + 1 \)
            else if \( \text{dist}[y] == \text{dist}[x] + \text{weight}(x, y) \) and \( \text{numedges}[x] + 1 < \text{numedges}[y] \)
                \( \text{prev}[y] = x \)
                \( \text{numedges}[y] = \text{numedges}[x] + 1 \)
        return \( \text{dist}[\cdot], \text{prev}[] \)

procedure path_from_s_to_target(prev[], target)
    Let \( S \) be a stack
    \( v = \text{target} \)
    while \( \text{prev}[v] \) is defined
        \( S.\text{push}(v) \)
        \( v = \text{prev}[v] \)
        \( S.\text{push}(v) \)
    return \( S \)
```

Runtime Complexity: \( \mathcal{O}((|V| + |E|) \log(V)) \). Clearly, the runtime complexity is same as that of
the Dijkstra’s algorithm as we are are just keeping track of numedges for each vertex which does not effect the runtime.

6. A directed graph $G = (V, E)$ is called semiconnected if for each pair of distinct vertices $u, v \in V$ there exists either a directed path from $u$ to $v$ in $G$, or a directed path from $v$ to $u$ in $G$ (or, possibly, both). Design an efficient algorithm to determine whether a given directed graph $G$ is semiconnected. Justify why your algorithm works, give pseudocode, and give an analysis of runtime complexity.

**Solution:** Let $G'$ be the DAG on the strongly connected components of $G$. Number the vertices of $G'$ in some topological order.

**Claim 1:** $G$ is semiconnected if there is always a path from the $i^{th}$ vertex to the $j^{th}$ vertex of $G'$ if $i < j$.

**Proof:** Suppose that there is always a path from the $i^{th}$ vertex to the $j^{th}$ vertex of $G'$ if $i < j$. Then, for any two vertices $u$ and $v$ in $G$, they either belong to the same SCC (in which case there is a path from $u$ to $v$ and a path from $v$ to $u$), or they belong to two different SCC’s, $c_u$ and $c_v$. Without loss of generality, suppose $c_u$ occurs earlier than $c_v$ with respect to the topological order. Then, by hypothesis, there is a path from $c_u$ to $c_v$, so there is a path from $u$ to $v$ as desired.

**Claim 2:** $G$ is not semiconnected if $\exists i, j$ such that $i < j$ and there is no path from the $i^{th}$ vertex to the $j^{th}$ vertex of $G'$.

**Proof** Suppose that for some topological sort of $G'$, there is no path from the $i^{th}$ vertex (with respect to this sort) to the $j^{th}$ vertex, where $i < j$. Then, there can be no path from the $j^{th}$ vertex to the $i^{th}$ vertex since $i < j$ and the vertices are topologically sorted. Hence, if $u$ is a vertex lying in the $i^{th}$ SCC and $v$ a vertex lying in the $j^{th}$ SCC, there are no paths from $u$ to $v$ or from $v$ to $u$, so $G$ is not semiconnected.

From claim 1 and claim 2 it follows that $G$ is semiconnected if and only if there is always a path from the $i^{th}$ vertex to the $j^{th}$ vertex of $G'$ if $i < j$.

This observation allows us to devise an algorithm to determine if $G$ is semiconnected: we first compute the strongly connected components of $G$ and construct the graph $G'$. Then, using DFS from any vertex in $G'$, we can topologically sort $G'$. Now, we check if there is a path from the $i^{th}$ SCC to the $j^{th}$ SCC if $i < j$. Note that this is the case if and only if there is an edge in $G'$ from the $i^{th}$ SCC to the $(i+1)^{th}$ SCC. Hence, we can just scan through the topological sort to see if the $i^{th}$ SCC has an edge to the $(i+1)^{th}$ SCC.

**Pseudocode**

```
procedure is_semiconnected(G)
    G' = get_metagraph(G)
    T[] = topological_sort(G') (linearize the meta vertices)
    for i = 0 to i < length(T)-1:
        if no edge from T[i] to T[i + 1] in G'
            return false
    return true
```
**Runtime Complexity** It takes $O(|V| + |E|)$ time for creating a metagraph and obtaining a topological ordering. Then we just need to do a scan through the topological ordering which is $O(|V|)$. Therefore, the total running time complexity of the algorithm is $O(|V| + |E|)$.

7. Let $dist[v]$ denote the shortest distance from the source vertex $s$ to $v$ in a given directed graph $G = (V, E)$. Design an efficient algorithm that sets $dist[v] = -\infty$ for all the vertices $v$ that can be reached from the source via a negative cycle. Justify why your algorithm works, give pseudocode, and give an analysis of runtime complexity.

**Solution:** We can determine whether a graph has a negative cycle using Bellman-Ford algorithm by checking if there is any update to the shortest distance to any vertex on the $|V|^\text{th}$ iteration. Since the graph has $|V|$ vertices, any path that uses $|V|$ edges must have touched some vertex twice because a path using $|V|$ distinct vertices would be $|V| + 1$ edges long. Therefore, if there is an update of the shortest distance for any vertex $v \in V$, then we can say “Negative Cycle Detected”.

It is important to see that if a negative cycle does exist in a graph, the shortest distance from the source at the $|V|^\text{th}$ iteration will not change for each vertex that can be reached from the source via a negative cycle. For some vertices that are reachable by a negative cycle, $|V|$ vertices are not enough to have traveled down a negative cycle.

To solve this problem, we run the Bellman Ford algorithm for $2|V|$ iterations. In up to $2|V|$ vertices, we can get from the source to any vertex $v$ as well as have enough edges to travel along any negative cycle in the graph. If there is a negative cycle in a path from the source to $v$, then the path itself without the negative cycle has up to $|V|$ vertices, while the negative cycle has up to $|V|$ vertices as well. Any negative cycle of more than $|V|$ vertices can be decomposed into two smaller cycles, at least one of which is a negative cycle.

Pseudocode

```
procedure set_distance(G, s)
for all $x \in V \setminus s$
    dist1[x] = $\infty$
    dist1[s] = 0
for $i = 1$ to $|V| - 1$
    for all $(x, y) \in G$
        if dist1[x] + weight(x, y) < dist1[y]:
            dist1[y] = dist1[x] + weight(x, y)
    for all $x \in V$
        dist2[x] = dist1[x]
    for $i = 1$ to $|V| + 1$
        for all $(x, y) \in G$
            if dist2[x] + weight(x, y) < dist2[y]:
                dist2[y] = dist2[x] + weight(x, y)
        for all $x \in V$
            if dist2[x] is not equal to dist1[x]:
                dist[x] = $-\infty$
            else:
                dist[x] = dist1[x]
return dist[]
```
**Runtime Complexity** The runtime complexity of this solution is still $O(|V||E|)$ because we are simply running the Bellman-Ford algorithm an extra $|V|$ iterations.