Exercises (strongly recommended, do not turn in)

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.

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.

![Graph Diagram]

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 $A$.

![Graph Diagram]

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 $A_1$ and $A_2$ of half the size. Does knowing the majority elements of $A_1$ and $A_2$ 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.)

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.

Problems (must be written up and turned in)

1. DPV 4.3: \textbf{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.

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.

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.

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 \( u, v \in V \), the \textit{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 \( u, v \in V \), finds a maximum bottleneck path from \( u \) to \( v \). Justify why your algorithm works, give pseudocode, and give an analysis of runtime complexity.

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.

6. A directed graph \( G = (V, E) \) is called \textit{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.
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.