Problem 1: Divide and Conquer Matrix Multiplication (Taken from DPV 2.27) [20 points].
If $A$ is a matrix, then $AA$ is the square of $A$.

(a) Show that five multiplications are sufficient to compute the square of a $2 \times 2$ matrix.

(b) What is wrong the the following algorithm for computing the square of an $n \times n$ matrix? Just use a divide-and-conquer approach as in Strassen’s algorithm except that instead of getting 7 subproblems of size $n/2$, we now get 5 subproblems of size $n/2$ by using our observation in part a. Using the analysis of Strassen’s algorithm, we get an algorithm for squaring a matrix that runs in $O(n^{\log_2(5)})$ time (note that $\log_2(5) \approx 2.32 < \log_2(7) \approx 2.81$, so this would be asymptotically faster than using Strassen’s algorithm).

(c) In fact, squaring matrices is no easier than matrix multiplication in general. Show that if you have an algorithm for squaring an $n \times n$ matrix in $O(n^c)$ time, then you can use it to multiply any two arbitrary $n \times n$ matrices in $O(n^c)$ time. [hint: Consider multiplying two matrices $A$ and $B$. Can you define a matrix whose square contains $AB$?]

Problem 2: Depth First Search [15 points]. In the textbook and in lecture, we examined a recursive implementation of the depth first search algorithm for an undirected graph $G = (V, E)$. The book claims that the same algorithm can be implemented using a stack instead of recursion (recall that a stack has two operations: (1) push an element to the top of the stack, and (2) pop an element from the top of the stack). Give an implementation of the depth first search algorithm that uses a stack instead of recursive calls. Explain why your algorithm has the same behavior as the recursive depth first search.

Problem 3: Bipartite Graph Search (Taken from DPV 3.7) [20 points]. A bipartite graph is a graph $G = (V, E)$ whose vertices can be partitioned into two sets $V_1$ and $V_2$ such that $V = V_1 \cup V_2$, $V_1 \cap V_2 = \emptyset$, and there are no edges between two vertices in the same partition.
(a) Give a linear time (in \(|V|\) and \(|E|\)) algorithm to determine whether an undirected graph \(G = (V, E)\) is bipartite. Argue for your algorithm’s correctness and its running time.

(b) Prove that an undirected graph is bipartite if and only if it contains no cycles of odd length (that is, a path beginning and ending at the same vertex with an odd number of edges).

**Problem 4: Counting Paths in Directed Acyclic Graphs [15 points].** Let \(G = (V, E)\) be a directed acyclic graph and consider two vertices \(s, t \in V\). Give an \(O(V + E)\) algorithm that counts the total number of different directed paths from \(s\) to \(t\) in \(G\). Explain why your algorithm is correct.

**Problem 5: Adapting Dijkstra’s algorithm [20 points].** Suppose we have a computer network represented as a directed graph \(G = (V, E)\) where vertices represent devices (routers, computers, etc) and edges represent connections from one device to another. Each edge \((u, v)\) has an associated weight \(p_{uv}\) that corresponds to the probability that a packet leaving device \(u\) will arrive at device \(v\) without being dropped. (Note that since \(p_{uv}\) is a probability, \(0 \leq p_{uv} \leq 1\)). Suppose that these probabilities are independent, so that a packet traveling edge \((u, v)\) and then \((v, w)\) has probability \(p_{uv}p_{vw}\) of arriving at \(w\) without being dropped, and so on for longer paths.

Suppose we want to route a packet from start device \(s\) to target device \(t\). Give an algorithm that finds a path from \(s\) to \(t\) that has maximum probability of arriving at \(t\) without being dropped. Your algorithm should have the same running time as Dijkstra’s algorithm. Explain why your algorithm is correct.