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 $AB$?]

Solution 1.

(a) Consider a $2 \times 2$ matrix $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$. We compute it’s square using standard matrix multiplication to get

$$AA = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}^2 + a_{12}a_{21} & a_{11}a_{12} + a_{12}a_{22} \\ a_{21}a_{11} + a_{22}a_{21} & a_{21}a_{12} + a_{22}^2 \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}^2 + a_{12}a_{21} & a_{12}(a_{11} + a_{22}) \\ a_{21}(a_{11} + a_{22}) & a_{21}a_{12} + a_{22}^2 \end{bmatrix}$$
Therefore, we only need to make the five unique multiplications appearing here: \(a_{11}^2, a_{22}^2, a_{12}a_{21}, a_{12}(a_{11} + a_{22}),\) and \(a_{21}(a_{11} + a_{22}).\)

(b) The problem with the suggested algorithm is that this trick only works at the top level of recursion. That is, some of the recursive calls will need to compute solutions to matrix multiplications that are not themselves problems of squaring a matrix, to which this trick cannot be applied. There is also a problem in that part a relied on the fact that regular multiplication is commutative, whereas matrix multiplication is not. Either is a fine answer.

(c) Suppose we have an algorithm for squaring an \(n \times n\) matrix in \(O(n^c)\) time. Let \(A\) and \(B\) be two \(n \times n\) matrices. Consider constructing the \(2n \times 2n\) matrix \(M = \begin{bmatrix} 0 & A \\ B & 0 \end{bmatrix}\) with the matrix \(A\) in the upper right quadrant, \(B\) in the lower left quadrant, and zeros everywhere else. Then we can run our squaring algorithm on \(M\) to compute \(MM\) in \(O((2n)^c) = O(2^cn^c) = O(n^c)\) time. But \(MM = \begin{bmatrix} AB & 0 \\ 0 & BA \end{bmatrix}\), so this implies we can compute \(AB\) in \(O(n^c)\) time by computing \(M^2\) and then reading off \(AB\) from the upper left quadrant of \(M^2\).

**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.

**Solution 2.** This implementation of DFS with a stack has the same behavior as the recursive implementation we gave in class.

```plaintext
1: procedure DFS(G)
2:   for u ∈ V do
3:     visit[u] = false
4:   end for
5:   t = 0
6:   Initialize empty stack S
7:   while ∃u ∈ V with visit[u] = false do
8:     Push(S, u)
9:     while S ≠ ∅ do
10:        v = Pop(S)
11:        pre[v] = t; t = t + 1
12:        for (v, t) ∈ E do
13:          if visit[t] = false then
14:            visit[t] = true
15:            Push(S, t)
16:          end if
17:        end for
18:        post[v] = t; t = t + 1
19:   end while
20: end procedure
```

The outer while loop is the same as in DFS, with the only difference being that rather than calling explore\((u)\), we instead push \(u\) onto a stack \(S\) (note that pushing to a stack is how function calls are actually implemented). Then, popping \(v\) off of the stack corresponds to looking at the most recent \(v\) in the sequence of recursive calls for which there were unexplored edges. Since we push all of the unexplored neighbors of
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).

Solution 3.

(a) Our algorithm will proceed by coloring the vertices red or blue during a breadth first search. In particular, choose an arbitrary starting vertex $s$ and color $s$ red. Run $BFS(G,s)$ to compute $d(u)$, the shortest path distance to $u$ from $s$, for every $u \in V$. Now, color all vertices at even distance from $s$ red, and color all vertices at odd distance from $s$ blue. The graph is bipartite if and only if there are no edges between two vertices of the same color. Because breadth first search has a running time of $O(|V| + |E|)$ and we are just adding a linear scan over the edges at the end, this algorithm also has a running time of $O(|V| + |E|)$.

To see that this algorithm is correct, suppose $G$ is bipartite. Without loss of generality, suppose $s \in V_1$. Then all of the neighbors of $s$ are in $V_2$, and the algorithm colors these vertices blue. All of the neighbors of these blue vertices must be in $V_1$, and so the algorithm colors them red, and so on. So at no point do we find an edge with both endpoints having the same color. Suppose instead that the algorithm does not find an edge with both endpoints the same color, but the graph is not bipartite. Then in particular the graph is not bipartite when we consider letting $V_1$ be all of the red vertices and $V_2$ be all of the blue vertices, so there must be an edge between two red vertices or two blue vertices, a contradiction.

(b) Let $G$ be an undirected graph. For one direction, suppose that $G$ is bipartite. Consider a cycle $C = (v_1, v_2, v_3, \ldots, v_{|C|-1}, v_1)$ in the graph. Suppose without loss of generality that $v_1 \in V_1$. Then, since $G$ is bipartite, $v_2 \in V_2, v_3 \in V_1, \ldots, v_{|C|} \in V_2$. But this is only possible if $|C|$ is even.

For the other direction, suppose that $G$ contains no odd cycles. Again run our coloring algorithm from part a, starting at an arbitrary vertex $s$. Let $R$ be the set of red vertices and let $B$ be the set of blue vertices. Note that $R \cap B = \emptyset$. Suppose for a contradiction that $r_1, r_2 \in R$ are adjacent (that is, there is an edge $(r_1, r_2)$). Since $r_1, r_2 \in R$, it must be that there is an even length path from $s$ to $r_1$ (call it $P(s,r_1)$) and similarly an even length path from $s$ to $r_2$ (call it $P(s,r_2)$). Let $P(r_2,s)$ be the path $P(s,r_2)$ taken in reverse. Then $P(s,r_1) - r_2 - P(r_2,s)$ is an odd length cycle, a contradiction.

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.

Solution 4. Let $n = |V|$. Since $G$ is a directed acyclic graph, we can linearize the graph (in linear time) so as to produce a labeling of the vertices $v_1, v_2, \ldots, v_n$ where all edges in the graph go from lower to higher labeled vertices (in fact, you can find these indices using post numbers from DFS as suggested on page 90 in the textbook). Assuming this has been done, our algorithm finds the total number of different directed paths from $v_i$ to $v_j$. To answer the question in terms of $s$ and $t$, just find the correct indices $i$ and $j$ such that $v_i = s$ and $v_j = t$, also in linear time. Also, note that without loss of generality, we can throw away any vertices that occur later in the order than $t$, since $t$ must be unreachable from any of these vertices.

The algorithm runs in linear time because it only considers each vertex and each edge at most once (note that the inner for loop is over edges starting at $v_k$, not all edges). To argue the correctness, we proceed
by induction. In the base case, when \( i = j \), the outer for loop is empty and we simply return 1, which is trivially correct.

Make the strong inductive hypothesis that for some \( k \leq j \), we have correctly computed the value of \( \text{paths}(l) \) for all \( k \leq l \leq j \). Then consider vertex \( v_{k-1} \) (note that the induction goes backward from \( j \) to \( i \)). Since the graph is a sorted DAG, all of the edges starting at \( v_{k-1} \) must go to some \( v_l \) with \( l \geq k \). Since each such edge is unique, it creates a unique path in combination with whatever paths originate at that child node. Since we have correctly computed these by the inductive hypothesis, the sum over the \( \text{paths}(l) \) of all of the children \( v_l \) of \( v_{k-1} \) is the value of \( v_{k-1} \), which is exactly what the algorithm does.

The problem can also be solved recursively rather than iteratively, using the acyclic property of a DAG to argue why the algorithm runs efficiently without cycling. Either solution is fine.

**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.

**Solution 5**. One solution is to transform all of the weights in the graph. In particular, let \( G' \) be the graph \( G \) where we replace each edge weight \( p_e \) with \( w_e = -\ln(p_e) \). Note that because \( 0 \leq p_e \leq 1 \), we know that \( -\ln(p_e) \) is nonnegative. Now, run Dijkstra’s algorithm on \( G' \) with start vertex \( s \) and target vertex \( t \). Dijkstra’s algorithm returns a shortest path \( P(s, t) \) in \( G' \). We want to prove that this is a maximum probability path in \( G \) from \( s \) to \( t \).
Since $P(s, t)$ is a shortest path in $G'$, we know that for any other path from $s$ to $t$ $P'(s, t)$

$$\sum_{e \in P(s, t)} w_e \leq \sum_{e \in P'(s, t)} w_e$$

$$\Rightarrow \sum_{e \in P(s, t)} - \ln(p_e) \leq \sum_{e \in P'(s, t)} - \ln(p_e)$$

$$\Rightarrow - \ln \left( \prod_{e \in P(s, t)} p_e \right) \leq - \ln \left( \prod_{e \in P'(s, t)} p_e \right)$$

$$\Rightarrow \ln \left( \prod_{e \in P(s, t)} p_e \right) \geq \ln \left( \prod_{e \in P'(s, t)} p_e \right)$$

$$\prod_{e \in P(s, t)} p_e \geq \prod_{e \in P'(s, t)} p_e$$

where the last inequality follows because log is a monotone increasing function for nonnegative numbers. So $P(s, t)$ must have maximum probability among all paths from $s$ to $t$, which is what we wanted to show.

It is also possible directly modify Dijkstra’s algorithm, and that solution will also be accepted. The pseudocode for such a modification is given below.

```
1: procedure Route(G, s, t)
2: Initialize Prob(s) = 1 and Prob(u) = 0 for all u ≠ s
3: Initialize priority queue Q all vertices in V, using Prob values for keys
4: while Q not empty do
5:   u = ExtractMax(Q)
6:   for (u, v) ∈ E do
7:     if Prob(v) < Prob(u) · p_uv then
8:       Prob(v) = Prob(u) · p_uv
9:       Parent(v) = u
10:      Update the key of v to Prob(v)
11:   end if
12: end for
13: end while
14: end procedure
```