Problem 1

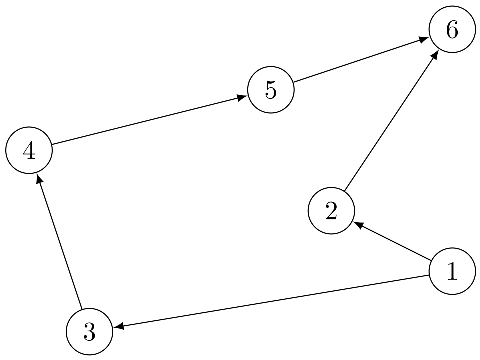
An ordered graph G = {V,E}, such that V = $\{v_1, v_2, ..., v_n\}$ is a directed graph with the following properties:

i) $e = \{(v_i, v_j)|i < j\}\ \forall e \in E$.

ii) $\delta_{out}(v) \geq 1\ \forall v_i, i = 1, 2, ..., n-1$.

Given an ordered graph G, find the length of the longest path that begins at v_1 and ends at v_n.

Counterexample for algorithm 3a:



In the figure above, the longest path is $\{v_1, v_3, v_4, v_5, v_6\}$, with length 4. The algorithm given returns $\{v_1, v_2, v_6\}$, which is a path of length $2 < 4$.

Solution: Design and Reasoning

We want a path which maximizes the number of edges, M, between nodes v_1 and v_n. Clearly, the last node and first node must be part of every solution, including the optimal solution. Suppose that an optimal solution O exists - then, we can characterize it by performing a case analysis and identifying potential sub-solutions: Since we know that $v_n \in O$, at least one of the nodes leading into v_n must also be in O. These $\delta_{in}(v_n)$ nodes form a subset $V_n \subset V$ and their edges terminating form a subset E_n. Following this pattern, we can break G into subgraphs $G_i = \{V_i, E_i\}$, where $V_i = \{v_j \in V|(v_j, v_i) \in E_{out}\}$ and $E_i = \{(u, v) \in E|v = v_i\}$ with sub-solutions of length l_i. If a node $v_i \notin O$, then the subset of edges incident on v_i is also not in O, so we can simply ignore those edges and continue searching. If $v_i \in O$, then we search for the sub-path of max length terminating at v_i in the subgraph G_i. Then, the max-length path terminating at v_i is M_i with length l_i, where $l_i = max(l_1, ..., l_{i-1}) + 1$.

Solution: Implementation

Like the majority of solutions seen in this chapter, this algorithm stores the max-length of paths terminating at each node in an array called L as it iterates through nodes. If we did this recursively, we would begin at the last node. However, we use an iterative solution, in keeping with the style of this chapter:

```
Let L be array of size n.
Initialize the array to 0.
For i in 2,...,n:
    For all j such that (v_j, v_i) ∈ E:
        If L[j] > L[i] − 1,
            L[i] = L[j] + 1.
Return L[n].
```

Solution: Proof and Time Complexity

The algorithm stores the max-length of paths terminating at a given node v_i. By definition, L[1] = 0. For i = 2, the algorithm correctly computes that L[2] = 1, since L[1] = 0 > -1 = L[2]-1. Let $i > 1$, and suppose by way of induction that the algorithm correctly computes L[i] for all $i < n$. Then, l_n = L[n] = $max(l_j, l_{j+1}, ..., l_{n-1}) + 1$, as desired. Since the i^{th} node has at most $i - 1$ in-edges, the inner loop runs at most $\sum_{i=1}^{n}(n - i) = \frac{n*(n-2)}{2}$ times. Thus, the algorithm is $\mathcal{O}(n^2)$.

Problem 2

The exercise asks us to write an algorithm that, given a sequence of n supply values (in lbs per week), returns an optimized schedule for the shipment of supplies at minimum overall cost. Cost is either a function of weight or time, depending on the shipping company chosen: Company A charges r dollars per lb, and company B charges c dollars per week for a 4-week (consecutive) contract.

Solution: Design and Reasoning

We want to find all 4-subsequences such that $4c < r\sum_{i=k}^{k+4} S_i$. Those are the subsequences that will be shipped via company B. Suppose there is an optimal sequence of companies O. Because the supply values must be shipped out sequentially, we know that if B is chosen in the i^{th} week in O, then the cost at the $i + 4^{th}$ point in time is $c_{i+4} = c_{i-1} + 4c$. Similarly, if A chosen in the i^{th} week, the cost at point i+4 is $c_{i+4} = rs_{i+4} + c_{i-3}$. Thus, when choosing company A or B for the i^{th} week in our schedule, we must minimize c_i such that $c_i = min(4c + c_{i-4}, rs_i + c_{i-1})$.

Solution: Implementation

```
Let C be an array of size n.
Let S be an array of size n.
Initialize C to 0.
For i in 1,...,n:
        C[i] = min(4c+C[i-4], rs_i+C[i-1]).
        If C[i] = 4c+C[i-4],
                S[i] = B.
        Else, C[i] = rs_i+C[i-1],
                S[i] = A.
Return S, C[n].
```

Solution: Time Complexity

Each iteration of the loop assigns a cost equal to $min(4c+c_{i-4}, rs_i+c_{i-1})$ for each week, as desired. The main loop runs each time, and each iteration requires a constant k steps to check for the appropriate minimum cost of the shipping s_i. Thus, the algorithm is $\mathcal{O}(n)$.

Problem 3

A rising trend in stock prices over n days is defined as a k-subsequence of the n prices over i_k days such that the following conditions hold:

(i). $i_1 = 1$.

(ii). $P_i < P_{i+1}$ for each j = 1, 2, ..., k-1.

Counterexample for algorithm 17a:

The algorithm fails on the subsequence 2, 4, 3, 4, 5, 7, 6, 7, returning 2, 4, 5, 7 with length 4 when the correct subsequence is 2, 3, 4, 5, 6, 7 with length 6.

Solution: Design and Reasoning

Suppose that there is an optimal solution O that returns length L_k for some longest k-subsequence. The subsequence ending at i_{k-1} has length L_{k-1} and consists of the longest rising trend observed from i_1 to i_{k-1}. Thus, $L_k = L_{k-1} + 1$. If a price $P_i \in L_k$, then $P_i < P_j \forall j > i$, thus any $P_{i-1} \in L_k$ must be smaller than P_i. For some P_i, we can store the longest subsequence from P_1 to P_i. Then, $L_{i+1} = L_i + 1$ if $P_{i+1} > P_i$. Else, $L_{i+1} = 1 + max(L_j) \forall j < i$.

Solution: Implementation

```
Let L be an array of size n, P an array of stock prices. Initialize L[1] = 0, L[i] = -1, i=2,...,n.
For i in 2,...,n:
    If P[i-1] ¡ P[i], L[i] = max(0, L[i-1]+1).
    Else, j = i-2, l = 0.
        While P[j] ¡ P[i] and j ¿ 1:
            l = max(L[j], l).
            j = j-1.
        L[i] = l.
Return max(L[1,...,n]).
```

Solution: Proof and Time Complexity

We can prove that this algorithm returns a correct solution by induction on i. For i=2, the algorithm returns 1 if $P_1 < P_2$ or else it returns 0. This is the correct length for a subsequence beginning at 1 and ending at 2. Assume that the algorithm returns the correct length for all stocks P_i from 1 to n-1. Then, for the n^th stock, $L[n] = L[n-1] + 1$ if P_n is larger than P_{n-1} or 1 + max-length for a subsequence of stocks between 1 and n-2, as desired. The main for loop runs n times, and each iteration i requires *i-2* iterations of the while loop, so this algorithm is $\mathcal{O}(n^2)$.

Problem 4

I'm unsure of how to approach this problem. I read the section in the textbook but still feel as though I don't truly understand what I'm being asked, let alone the solution. Hopefully the TA can go over this type of problem during discussion, since it was not covered in lecture.

Problem 5

Description

The exercise asks us to produce an efficient algorithm that computes the number of shortest paths between (v,w) in a weighted graph G with no negative cycles (but possibly negative edge weights).

Solution: Reasoning and Implementation

Since we have a weighted graph, we can't use BFS, and since we have negative edge weights, we can't use Dijkstra's algorithm. Thus, we must use a modified version of the Bellman-Ford algorithm described in [KT], section 6.9. First, we have to compute the length of the shortest path, which we can do using the algorithm described on [KT] pg 294. Then, we can add an additional for-loop to the algorithm to compute the number of shortest paths from v to w with the desired length.

Let n = number of nodes in G.
Array M[0,...,n-1, V].
Define M[k, v] = 0, M[k, u]=, if u $\neq$ v.
For i in 1,...,n-1:
    For j in 1,...,n:
        M[i, j] = min(M[i-1, j-1] + c_{ij}).
P = min_1^i(M[i,w]), L=length(P), N=0.
    For i in 1,...,n:
        If M[P, i] $\leq$ P, N++.
Return N.

Solution: Time Complexity

The algorithm works in almost exactly the same way as the Bellman-Ford algorithm described in the textbook, with a few modifications that allow us to keep track of the number of shortest paths. The time complexity is $\mathcal{O}(n^3)$, since the inner most for-loop runs n^2 times at worst and examines up to n nodes per iteration during construction of the M matrix.