Question 1 (Strongly Connected Components, 15 points). Compute the strongly connected components of the graph below:

Running DFS on the reverse graph we get:

Running explore on the largest post number, C, we find the SCC CDFGI. Next we run explore on the largest remaining post number, H, yielding just H. Finally, we explore A, yielding the last SCC, ABEJ. Thus the SCCs are ABEJ, CDFGI, and H.
**Question 2** (Knapsack, 20 points). Compute the collection of distinct items from the list below with total weight at most 10, and total value as large as possible subject to this constraint.

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>D</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>E</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>F</td>
<td>6</td>
<td>14</td>
</tr>
</tbody>
</table>

We run our dynamic program to do this computation. We obtain the following table for the best value achievable using the noted subset of items and the capacity limit:

<table>
<thead>
<tr>
<th>Allowed Items</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>AB</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>ABC</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>11</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>ABCD</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>ABCDE</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>11</td>
<td>12</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>22</td>
</tr>
<tr>
<td>ABCDEF</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>11</td>
<td>14</td>
<td>15</td>
<td>19</td>
<td>20</td>
<td>22</td>
</tr>
</tbody>
</table>

Tracing things backwards, we find that the only way to get the 22 in the lower right, is by not using F (using F one could only get 14 + 7 = 21). This 22 could only be obtained by using E, and the best combination employing ABCD with total at most 5 (if we don’t use E, we can only get 19). This sum should not use D as then the best total would be 7 + 1 = 8, but it should use C and B. Thus, the optimal solution is BCE for a total value of 22.
**Question 3** (Subsequence Decision, 15 points). Given an algorithm that given two sequences \(a_1, \ldots, a_m\) and \(b_1, \ldots, b_n\) determines whether or not the \(a\)'s are a subsequence of the \(b\)'s. In particular, determine whether or not there is \(x_1 < x_2 < \ldots < x_m\) so that \(b_{x_i} = a_i\) for all \(i\).

For full credit, your algorithm should run in time \(O(n)\) or better.

We can solve this with a greedy algorithm. The key insight is that if the \(a\)'s are a subsequence of the \(b\)'s we can make them show up as early as possible. In particular, we can take \(x_{i+1}\) to be the smallest index \(k > x_i\) so that \(b_k = a_{i+1}\). This can be proven by an exchange argument. If we have a valid sequence of \(x\)'s where the first \(t\) satisfy this property, replacing \(x_{t+1}\) be the smallest index greater than \(x_t\) that stores the symbol \(a_{t+1}\) will maintain this property. Repeating this, we obtain a sequence that satisfies this for all \(t\).

Determining whether \(a\) is a subsequence of \(b\) in this way is quite easy. We simply scan through the \(b\)'s keeping track of how many of the \(a\)'s have been matched thus far. The algorithm is as follows:

```plaintext
SubsequenceDetection(a,b)
    i = 1  \(\text{\\ The next a that needs to be matched}\)
    for j = 1 to n  \(\text{\\ The location in the b sequence}\)
        if b[j] = a[i]
            i++
        if i>n
            return True
    return False
```

This clearly takes \(O(n)\) time.
Question 4 (Multiple Order Statistics, 15 points). In the multiple order statistics problem, you are given a list \( L \) of length \( n \) and given an integer \( m \) dividing \( n \) and asked to find the \((n/m)\text{th}, (2n/m)\text{th}, \ldots, ((m-1)n/m)\text{th} \) smallest elements of \( L \). For example if \( n = 6 \) and \( m = 2 \), you should return both the 3\text{rd} smallest element of \( L \) and the largest element of \( L \).

For full credit, your algorithm should run in time \( O(n \log(m)) \) or better.

The basic strategy will be by divide and conquer. Letting \( k \) be approximately \( m/2 \) we find that \((kn/m)\text{th}\) largest element and split the list into two pieces based on whether they are larger or smaller than this. We then need the \( k \) MOS of the first list and the \( m-k \) from the latter. The algorithm is as follows:

\[
\text{MOS}(L,m)
\]
\[
\text{if } m=1
\]
\[
\text{Let } x = \text{max}(L)
\]
\[
\text{return } \{x\}
\]
\[
\text{Let } k = \lfloor m/2 \rfloor
\]
\[
\text{Let } x = \text{OrderStats}(L, kn/m)
\]
\[
\text{Sort } L \text{ into:}
\]
\[
\text{A (the elements } \leq x \text{) and}
\]
\[
\text{B (the elements } > x \text{)}
\]
\[
\text{return MOS(A, k) U MOS(B, m-k)}
\]

To analyze the runtime, note that we have parameters both \( m \) and \( n \). We have that \( T(m, n) \) is \( O(n) \) if \( m = 1 \). Otherwise, we spend \( O(n) \) time finding \( x \) and splitting up our list, and then need to solve two subproblems with both \( n \) and \( m \) half as large.

\[
T(n, m) = 2T(n/2, m/2) + O(n).
\]

We note that \( n/m \) remains constant throughout this algorithm, so calling this \( D \), we note that we have the recursion

\[
T(m) = 2T(m/2) + O(m).
\]

Using the Master Theorem, we note that we are in the balanced case where \( a = b^d \), so the runtime is \( T(m) = O(Dm \log(m)) = O(n \log(m)) \). [The extra factor of \( D \) is there because it was an extra factor in the additive term in each step.]
Question 5 (Road Trip, 15 points). Karkat is planning a road trip. He has a map of the country given by a weighted, undirected graph $G = (V, E)$ where the (positive) weights are the lengths of the roads. He is trying to get from vertex $s$ to vertex $t$. However, his car has a limited gas tank and can only travel at most $m$ miles on a single tank. However, there are $k > 0$ vertices of $G$ that are labelled as gas stations, where he can refill his tank. In other words, Karkat needs to find a path from $s$ to $t$ that has length at most $m$ between gas stations. Give an algorithm that given $G, m$ and the set of gas stations, finds the shortest such path.

For full credit, your algorithm should run in time $O(k(|V| + |E|) \log(|V|))$ or better.

First, we note that adding gas stations at $s$ and $t$ does not change the problem. We then produce a graph on gas stations where there is an edge between two stations if Karkat can get from one to the other without refuelling in between, where the edge length is the length of the path. The question then boils down to finding the shortest path from $s$ to $t$ in this new graph.

The algorithm is as follows:

RoadTrip

Let $S$ be the set of vertices that are $s$ or $t$ or a gas station
Create a new graph $G'$ with vertices given by elements of $S$
for each $v$ in $S$
    run Dijkstra on $G$ starting at $v$ to find shortest paths to other vertices
for each $w$ in $S$
    if $\text{dist}(v, w) \leq m$
          add an edge from $v$ to $w$ in $G$ of length $\text{dist}(v, w)$
    run Dijkstra on $G'$ and return the distance from $s$ to $t$

The runtime is not difficult to compute. We run Dijkstra on $G$ once for each element of $S$ ($O(k)$ times for a total time of $O(k(|V| + |E|) \log(|V|))$). We check at most $k^2 \leq k|V|$ pairs of edges for being in $G'$, and run Dijkstra on $G'$ once with runtime at most $O(k^2 \log(k)) = O(k(|V| + |E|) \log(|V|))$. 

Question 6 (Exact Path Length, 20 points). Given a DAG $G$ with edge weights (given as potentially large integers) and two specified vertices $s$ and $t$, the Exact Path Length problem asks whether or not there exists a path from $s$ to $t$ in $G$ with path length exactly equal to some given integer $N$. Prove that the Exact Path Length problem is NP-Complete.

We first note that this problem is in NP. If there is such a path, it is easy to verify that the path goes from $s$ to $t$ and has the correct length. We have left to show completeness.

We do this by reduction from Subset-Sum. In particular, for a set $S = \{x_1, x_2, \ldots, x_n\}$ and $N$ we consider the problem of whether or not some subset of $S$ sums to $N$. We show that we can produce an equivalent instance of Exact Path Length. In particular, we produce a graph with vertices $s = v_1, v_2, \ldots, v_{n+1} = t$. Between $v_i$ and $v_{i+1}$, we have two edges. One of weight 0 and the other of weight $x_i$. We note that $G$ is a DAG, since any path can only lead from $v_a$ to $v_b$ with $b > a$. Next, we claim that the lengths of the $s \to t$ paths in $G$ have lengths exactly equal to the sums of subsets of $S$. In particular, the lengths of the edges must be distinct elements of $S$, and so the total length is some subset sum. Conversely, if $T \subset S$, then we can construct a path in $G$ that uses the edges with lengths in $T$, and the length 0 edges elsewhere. Thus, $S$ has a subset summing to $N$ if and only if $G$ has an $s \to t$ path of length exactly $N$. 