1. As discussed in class, if \( f_i(y) \) is the optimal profit for \( \text{KNAP}(1, j, y) \), the recurrence relation for \( f_i(y) \) is given by: 
   \[
   f_i(y) = \max\{f_{i-1}(y), f_{i-1}(y - w_i) + p_i\}.
   \]
   Also, \( f_0(y) = 0 \) for all non-negative values of \( y \) and \( f_i(y) = -\infty \) when \( y \) is negative. From these relations we compute \( f_0(y), f_1(y), f_2(y), f_3(y), f_4(y) \) for all \( 0 \leq y \leq 5 \). These values are shown in the following table.

<table>
<thead>
<tr>
<th>Function</th>
<th>( y = 0 )</th>
<th>( y = 1 )</th>
<th>( y = 2 )</th>
<th>( y = 3 )</th>
<th>( y = 4 )</th>
<th>( y = 5 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_0 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( f_1 )</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>( f_2 )</td>
<td>0</td>
<td>10</td>
<td>15</td>
<td>25</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>( f_3 )</td>
<td>0</td>
<td>10</td>
<td>15</td>
<td>25</td>
<td>35</td>
<td>40</td>
</tr>
<tr>
<td>( f_4 )</td>
<td>0</td>
<td>10</td>
<td>15</td>
<td>25</td>
<td>35</td>
<td>40</td>
</tr>
</tbody>
</table>

For example, \( f_3(5) = \max\{f_2(5), f_2(5 - 3) + 25\} = \max\{25, 15 + 25\} = 40 \). Also, \( f_4(4) = \max\{f_3(4), f_3(4 - 2) + 12\} = \max\{35, 15 + 12\} = 35 \); and so on. Thus the optimal profit is 40.

2. Let \( A \) be the adjacency matrix of the graph (whose diagonal elements are zeros). It can be shown that \( A^k(i, j) = 1 \) iff there is a path from node \( i \) to node \( j \) of length exactly equal to \( k \), for any \( 0 \leq k \leq (n - 1) \). If there is a path at all from node \( i \) to node \( j \) in \( G \), the shortest such path will be of length \( \leq (n - 1) \).

Hence, \( A^n = I + A + A^2 + \ldots + A^{n-1} = (I + A)^{n-1} \). Here, scalar addition corresponds to boolean or and scalar multiplication corresponds to boolean and.

Now, \( (I + A)^{n-1} \) can be computed by repeated squaring, i.e., \( (I + A)^2, (I + A)^4, (I + A)^8 \) etc.

Complexity = Complexity of adding matrices \( I \) and \( A \) + Complexity of computing \( (I + A)^{n-1} = O(M(n) \log n) \).

3. Start from the first header node in the adjacency list of \( G \) and start counting the edges until you reach the count \( 2(|V| - 1) \). If the graph has any more edges than these then it is not a tree. If the graph has exactly \( 2(|V| - 1) \) edges, do the following: Perform a DFS in \( G \) and identify the connected components of \( G \). If \( G \) contains only one connected component,
then it is a tree else it is not a tree. Time Complexity = Initial edge counting time + Time complexity of DFS in G. DFS takes time $O(|V| + |E|) = O(|V|)$. Initial edge counting also takes $O(|V|)$ time.

4. It was shown in class that the maximum of $n$ elements can be found in $O(1)$ time using $n^2$ common CRCW PRAM processors.

Consider the case when $\epsilon = \frac{1}{2}$. Divide the elements into groups of size $\sqrt{n}$. Assign the first $\sqrt{n}$ elements to the first $n$ processors and the second $\sqrt{n}$ elements to the next $n$ processors and so on. The maximum element in each group can be found in $O(1)$ time. At this stage, we have $\sqrt{n}$ elements and $n\sqrt{n}$ processors. Hence, the maximum of these elements can be found in $O(1)$ time. Total time = $O(1)$.

Next, consider the case when $\epsilon = \frac{1}{3}$. Here, divide the elements into groups of size $n^{1/3}$. Assign the first $n^{1/3}$ elements to the first $n^{2/3}$ processors and the second $n^{1/3}$ elements to the next $n^{2/3}$ processors and so on. The maximum element of each group can be found in $O(1)$ time and using $n^{4/3}$ processors the maximum of these maximum elements can be found in $O(1)$ time.

For the general case, partition the input into groups with $n^\epsilon$ elements in each group. Find the maximum of each group assigning $n^{2\epsilon}$ processors to each group. This takes $O(1)$ time. Now the problem reduces to finding the maximum of $n^{1-\epsilon}$ elements. Again, partition the elements with $n^\epsilon$ elements in each group and find the maximum of each group. There will be only $n^{1-2\epsilon}$ elements left. Proceed in a similar fashion until the number of remaining elements is $\leq \sqrt{n}$. The maximum of these can be found in $O(1)$ time. Clearly, the run time of this algorithm is $O(1/\epsilon)$. This will be a constant if $\epsilon$ is a constant.

5. Let $k_1, k_2, \ldots, k_n$ be the elements. Divide the elements into groups of size $\log n$. Assign the first $\log n$ elements to the first processor and the second $\log n$ elements to the second processor and so on.

Create a temporary array, $A$, of size $n$.

Step 1: At each processor do: compare all the elements assigned to that processor with $x$ sequentially. If an element $k_i$ is less than or equal to $x$, place 1 in the array $A$ at index $i$ else place a 0 at $A_i$. Time = $O(\log n)$.

Step 2: Compute the prefix sums of the elements of the array $A$ using all the $\frac{n}{\log n}$ processors. Complexity = $O(\log n)$. 

2
Let us assume that the rearranged elements will be placed in an array \( B \).

Step 3: Move the elements that are less than or equal to \( x \) into \( B \) first.

For each element \( k_i \) with a 1 in \( A_i \), the value of the corresponding prefix sum gives the index in \( B \) where that element can be placed.

At each processor do: let \( p \) be the prefix sum of an element \( k_i \) with a 1 in \( A_i \). Move \( k_i \) into \( B[p] \). Time = \( O(\log n) \) (there are only \( \log n \) elements at each processor).

Step 3: Find the maximum prefix (\( \text{maxPrefix} \)) of any element with a 1 in \( A_i \) in step 2.

Step 4: Now, the elements that are greater than \( x \) can be copied into \( B \) starting from \( B[\text{maxPrefix} + 1] \).

Flip all the elements of \( A \) and repeat the procedure in steps 2 and 3 with the following difference: if \( p \) is the prefix sum of an element \( k_i \), move \( k_i \) into \( B[p + \text{maxPrefix}] \). Complexity = \( O(\log n) \).

Total complexity = \( O(\log n) \).

6. We know that \( \pi_1 \) polynomially reduces to \( \pi_2 \). Let \( x \) be an instance of \( \pi_1 \) with \( |x| = n \). We can convert this into an instance \( x' \) of \( \pi_2 \) in \( O(n^c) \) time (for some constant \( c \)). Note that \( c \) could be any constant (10, for instance) and we can only say that \( |x'| = O(n^c) \) and in fact \( |x'| \) could be \( \Omega(n^c) \). If \( |x'| \) is \( \Omega(n^c) \), the run time needed for solving \( x' \) will be \( O(2^{\sqrt{\Omega(n^c)}}) \) which can be asymptotically greater than \( 2^{\sqrt{n}} \). Thus the given statement is not correct.

7. Use the following algorithm, \textbf{Size}(\textit{Graph} \ G) -

\begin{verbatim}
for \( i := |V| \) to 0 do
  if \( \text{CLQ}(i) = \text{yes} \) then
    output \( i \)
    quit
  end
end
\end{verbatim}

Note that we increase the runtime of the \textbf{CLQ} algorithm, by a factor of \( |V| \), yet maintaining it polynomial.