1. One possible greedy algorithm will sort the programs in nondecreasing order of their tape requirements and pick the \( r \) smallest programs whose total length is \( \leq l \) and the total length of the \( (r+1) \) smallest programs is \( > l \). The run time of this algorithm is \( O(n \log n) \).

Let \( R = \{R_1, \ldots, R_r\} \) be the output of greedy algorithm, and \( S = \{S_1, \ldots, S_s\} \) be that of an optimal strategy in nondecreasing order of program size. Let \( i \) be the least index in which these two differ. Clearly, \( R_i < S_i \). Also note that \( R_i \not\in S \). Thus we can insert \( R_i \) into \( S \) before \( S_i \) and delete \( S_s \), if there is a need. The new solution has at least the same number of programs as before. Proceeding in this fashion, we can make the first \( r \) programs of \( S \) the same as the corresponding \( r \) programs of \( R \). There can not be space left in the tape for any more programs. If there were, the greedy algorithm would have added at least one more program. Thus \( s \leq r \).

If each \( a_i > l \), the utilization ratio is 0.

2. (a). \( f_i(y) = \max\{f_{i-1}(y), \max_{k,y \geq kw_i}\{f_{i-1}(y - kw_i) + kp_i\}\} \).

(b). If we fix \( i \) and \( y \), \( f_i(y) \) can be computed in \( O(m) \) time. But \( i \in [1..n] \), \( y \in [1..m] \). Thus the running time is \( O(m^2n) \). Note that we can bring the run time down to \( O(mn) \).

(c). For \( i \in [1..n] \), calculate \( p_i/w_i \). As a result, find the object \( k \) with the maximum profit density. Fill the knapsack with this object. Then \( x_k = m/w_k \), and the total profit is \( \frac{m}{w_k} p_k \).

3. Let \( \text{minCount}(x) \) return the minimum number of coins for the amount \( x \).

\[
\text{minCount}(x) = \min\{\text{minCount}(x - a_1) + 1, \text{minCount}(x - a_2) + 1, \ldots, \text{minCount}(x - a_n) + 1\}
\]

By computing \( \text{minCount}(x) \) for each possible value of \( x, 1 \leq x \leq C \) and storing them in a table, we can see that the value of \( \text{minCount} \) for each value can be derived from the \( \text{minCount} \) values of \( n \) other entries. Complexity = \( O(Cn) \).

4. If a graph contains a square as a subgraph then there exist at least two nodes which have two common neighbors.

Step 1: Arrange the neighbors of each node in a sorted order. This can be done by constructing tuples \((i,j)\) for each neighbor \( j \) of a node \( i \) and sorting all the constructed tuples of all the nodes using the radix sort. This takes \( O(|V| + |E|) \) time.

Step 2: Find if there are any two nodes that have two common neighbors.

Let \( L_i \) represent the sorted list of neighbors of node \( i \), \( 1 \leq i \leq n \).

- for \( i := 1 \) to \(|V|\) do
  - for \( j := 1 \) to \(|V|\) do
    - Merge \( L_i \) and \( L_j \) and check whether \( i \) and \( j \) have two common neighbors.

Run time of step 1 is \( O(|V| + |E|) \).

Let \( d_i \) represent the length of \( L_i \). Note that \( d_i \) is the degree of \( i \).

Run time of step 2 is \( \sum_{1 \leq i \leq |V|, 1 \leq j \leq |V|} (d_i + d_j) = O(|V||E|) \).

Thus the total run time of the algorithm is \( O(|V||E|) \).

5. Evaluate \( f(a) \) and \( g(a) \) using Horner’s rule and multiply \( f(a) \) and \( g(a) \). Complexity = \( O(n) \).

6. Sort \( A \) and \( B \). Let \( a_i \) be the number of elements \( x \in A \) such that \( x = i \). Let \( b_i \) be the number of elements \( y \in B \) such that \( y = i \). Let \( P_1(x) = a_0x^{5n} + \ldots + a_0, P_2(x) = b_0x^{5n} + \ldots + b_0 \). Let \( P_3(x) = P_1(x)P_2(x) = c_0x^{10n} + \ldots + c_0 \). Then \( c_i = |C_i| \). Sorting \( A \) and \( B \) takes \( O(n) \) time, computing \( a_i, b_i \) for \( i = 0, \ldots, 5n \) takes \( O(n) \) time, and computing \( c_i \) for \( i = 0, \ldots, 10n \) takes \( O(n \log n) \) time.