**Question 1 – Coin change revisited.**

**Part (A)**
In the naive dynamic programming algorithm you compute minimum number of the coins for N cents as

\[ M(N) = \min (M(N[d_1]), M(N[d_2]), ..., M(N[d_k])) + 1 \]

We initialize \( M(i) = 0 \) for all \( i < \min(d_i) \), and compute \( M(i)s \), one by one all the way through \( M(N) \). This algorithm uses \( O(N) \) memory.

If we look closely in the dynamic programming algorithm, we will see for computing each \( M(i) \) we need at most the last \( \max(d_i) \) values. So using the same algorithm we only keep the last \( \max(d_i) \) values and get rid of the rest.

**Part (B)**
In this part we first sort the coins \( (d_1 \leq ... \leq d_k) \) and use a dynamic programming for computing \( D(n, i) \) which shows the number of the different ways to obtain \( n \) cents using first \( i \) coins. In each step we decide if we should use a \( d_i \) cent coin or not using this formula:

\[ D(n, i) = D(n - d_i, i) + D(n, i - 1) \]

**Proof of correctness:** By induction over \( i \) we can prove that we are counting every possible way for obtaining \( n \) cents. Also we need to prove that we only count each one of them at most once. Again by induction we show every way in \( D(n, i - 1) \) set is unique as well as every way in \( D(n - d_i, i) \) set. So if we add a \( d_i \) cent coin to all of \( D(n - d_i, i) \) they’ll still be unique. Also the two sets don’t overlap, since in one of them we have \( d_i \) cent coins and in the other we don’t.

**Proof of time:** For computing \( D(N, k) \) there are \( N \times k \) elements that needed to be returned and computing each one takes \( O(1) \), so the algorithm is polynomial in \( N, k \).

**Question 2 – Counting paths.**

**Part (A)**
Let \( A \) be the adjacency matrix of the graph \( G(V, E) \). Number of \( k \) length paths between \( u \) and \( v \) is given by \( [u, v]^{th} \) entry in the matrix \( A^k \).

**Correctness:** \( A^k \) gives the number of total \( k \) length walks from vertex \( u \) to vertex \( v \). Since this is a DAG, there will be no repeated vertices in a walk between vertices \( u \) and \( v \).

**Runtime:** Multiplying 2 square matrices takes \( O(n^3) \). Therefore, total runtime is \( O(kn^3) \).

**Part (B)**
Dynamic programming algorithms, including the above, fail because paths are defined not to have repeated vertices, and the algorithm would count walks that loop around as paths, although they are not.

**Question 3 – The ill-prepared burglar.**

**Part (A)** Let \( S = 10 \) and (value, size) pairs be \((7, 6), (5, 5), (5, 5)\).

**Part (B)** Same as above part.

**Question 4 – Road tripping.**

**Part (A)**
Example for this part would be 4 songs with durations 40, 10, 50, 20. The algorithm first write song 1 on CD1. Then song 2 on CD 1. For song 3 there is not enough space in CD1, so it writes
the song on CD2. For song 4 there is not enough space in CD1 and CD2. So it writes song 3 on CD3.
But the optimal solution uses 2 CDs. One for songs 1 and 4, and the other for songs 2 and 3.

**Part (B)**
We know all of CDs are at least half full (except at most one). Let’s say the first CD that is less than full is CD \#i. So all the next CDs should be more than half full. Because if they don’t they could fit in CD \#i. So summation of duration of of songs is more than 1/2 duration of CDs in greedy algorithm (SOL). And duration of optimal CDs (OPT) is less than duration of songs.

\[ \text{SOL}/2 \leq \sum d_i \leq \text{OPT} \]

**Question 5 – Maximizing happiness.**

**Part (A)**
Consider three gifts x, y, z and children 1,2,3. Suppose that child 1 has happiness values (1, 0, 2) to the gifts (x, y, z) respectively, child 2 has happiness values (2, 1, 0) and child 3 has values (0, 2, 1). Now, the solution x → 1, y → 2, z → 3 is locally optimal( because it cannot be strictly improved by any swap) and has a total happiness of 3. On the other hand, the solution z → 1, x → 2, y → 3 has a total happiness value equal to 6.

**Part (B)**
Consider a solution that is locally optimum under such ”Triple swaps”. Without loss of generality (by appropriately relabeling the gifts), we may assume that in this solution, gift 1 is given to child 1, gift 2 to child 2, and so on.
Let the optimal assignment be defined by \( \tau \), a mapping from [n] to itself. i.e., the optimal assignment gives gift i to child \( \tau(i) \).
Let \( i \) be any index. By local optimality (and \( A_{i,j} \geq 0 \)), we have
\[
A_{i,i} + A_{\tau(i),\tau(i)} + A_{\tau(\tau(i)),\tau(\tau(i))} \geq A_{i,\tau(i)} + A_{\tau(i),\tau(\tau(i))}
\]
(else a swap improves the solution)
Let us sum this over \( i \). The LHS is precisely \( 3 \cdot \sum_i A_{i,i} \), i.e., three times the total happiness of the locally optimum solution (as \( \tau() \) is a permutation). The RHS is \( 2 \cdot \sum_i A_{i,\tau(i)} \), i.e., twice the total happiness of the optimum solution. This immediately gives what we need to show.

**Question 6 – Uniqueness of spanning trees.**

**Part (A)**
Every two components should be connected with only the minimum edge between them. Because if they don’t, we can replace that edge with the minimum edge and achieve a spanning tree with less cost. Since edges are unique, then we will have a unique MST for every graph with unique edges.

**Part (B)**
Same reasoning applies here. For every two connected component we have only one edge connecting them with minimum edge. Because if we have two, then there would be a loop consisting of two edges with the same weight and it’s a contradiction.

**Question 7 – Reverse greedy.**

We prove that this process returns exactly the MST that Kruskal’s algorithm returns. First we show that it returns a spanning tree. Since it only removes edges if they don’t make the graph disconnected and it removes until the are no more loops.
Now we show that the edges that they pick are the same. For each edge \((e)\) in Kruskal’s algorithm, consider the two components that it connects. Our algorithm removes every other edge between these two components, since if their weight is greater than \((e)\), then algorithm reaches them first and remove them since they are part of a loop containing \((e)\). If they have the same weight, as question mentions they pick the same edge. And for edge \((e)\) our algorithm keeps it, since there is no more edge between these two components and by removing \((e)\), we are disconnecting the graph. So the result is the same MST as Kruskal’s algorithm output.