Problem 1:

Details regarding the dynamic programming table: Each segment \(\langle v_1, v_2, \ldots, v_i \rangle, 1 \leq i \leq n\), of the path constitutes a subproblem. We use a 1-dimensional array \(P[1..n]\) where for \(1 \leq i \leq n\), \(P[i]\) stores the minimum cost of a feasible placement of inspection stations for the segment \((v_1, v_2, \ldots, v_i)\), including inspection stations at \(v_1\) and \(v_i\).

Once we compute the values of all the elements of the array \(P\), the solution to the problem is the value of \(P[n]\).

Computation of the elements of array \(P\): For \(1 \leq x \leq y \leq n\), we use \(d(v_x, v_y)\) to denote the distance between nodes \(v_x\) and \(v_y\).

- Since \(c(v_1) = 0\), we have \(P[1] = 0\).
- Suppose we have computed the entries \(P[1..i-1]\) for some \(i \geq 2\). The reasoning for computing \(P[i]\), \(2 \leq i \leq n\), is as follows. Bear in mind that we can always place inspection stations at \(v_1\) and \(v_n\) (since \(c(v_1) = c(v_n) = 0\)) and that the definition of \(P[i]\) requires the placement of an inspection station at node \(v_i\).

(a) Since \(i \geq 2\), there is at least one inspection station that precedes \(v_i\).

(b) In any optimal placement for the segment \(\langle v_1, v_2, \ldots, v_i \rangle\), the inspection station that immediately precedes the one at \(v_i\) must be at a distance of at most \(\Gamma\) from \(v_i\). Thus, we need to consider only those nodes \(v_j\), \(1 \leq j \leq i-1\), for which \(d(v_j, v_i)\) is \(\leq \Gamma\) as candidates for the inspection station that immediately precedes the one at \(v_i\). Therefore, for \(i \geq 2\), the equation for computing \(P[i]\) is as follows:

\[
P[i] = c(v_i) + \min_{1 \leq j \leq i-1} \{P[j] : d(v_j, v_i) \leq \Gamma\}.
\]  

Steps of the Algorithm:

1. \(P[1] = 0\).

2. for \(i = 2\) to \(n\) do
   (a) \(\text{Temp} = \infty\). /* Temp is used to compute the minimum value. */
   (b) \(j = i - 1\).
   (c) while \((j \geq 1 \text{ and } d(v_j, v_i) \leq \Gamma)\) do
      (i) if \((P[j] < \text{Temp})\) then \(\text{Temp} = P[j]\).
      (ii) \(j = j - 1\).
   (d) \(P[i] = c(v_i) + \text{Temp}\).

3. Output \(P[n]\).
Running time: Steps 1 and 3 use $O(1)$ time. So, we focus on the time for Step 2.

To discuss an efficient implementation of Step 2, we first show that with $O(n)$ additional storage and preprocessing time, we can determine $d(v_i, v_j)$ for any $i \leq j$ in $O(1)$ time. We use an array $D[1..n]$, where $D[i]$ contains the value $d(v_1, v_i)$, $1 \leq i \leq n$. All the entries of $D$ can be computed in $O(n)$ time using the following equations. (Recall that the distance values $d(v_{i-1}, v_i)$, for $2 \leq i \leq n$, are given as part of the problem.)

$$D[1] = 0$$
$$D[i] = D[i-1] + d(v_{i-1}, v_i) \text{ for } 2 \leq i \leq n.$$

Once all the entries of $D$ are available, for any $i \leq j$, $d(v_i, v_j)$ can be computed in $O(1)$ time since $d(v_i, v_j) = D[j] - D[i]$.

We now discuss the running time of Step 2. The loop in Step 2 runs $n-1$ times. Each time through the loop, one entry $P[i]$ gets computed. This involves computing a minimum value over the previous $i-1 = O(n)$ values. The value of $P[i]$ can be computed in $O(n)$ time, since as observed above, the distance $d(v_j, v_i)$ can be computed in $O(1)$ time for any $j \leq i$. Therefore, the computation of all the entries of $P$ can be done in $O(n^2)$ time. In other words, the running time of the algorithm is $O(n^2)$.

Space requirement: The algorithm uses arrays $P$ and $D$, each with $n$ elements. So, the space required by the algorithm is $O(n)$.

---

**Problem 2:**

Idea: We use another array $S[1..n]$. At the end of the divide-and-conquer algorithm, element $S[i]$ of this array stores the length of a longest ascending subsequence starting at the element $A[i]$, $1 \leq i \leq n$. The final answer to the problem is the largest value in $S$.

Part (a):

**Divide Step:** Divide the array $S[1..n]$ into two parts, namely $S[1]$ and $S[2..n]$. This step runs in $O(1)$ time. (Note that the two parts are not necessarily of the same size.)

**Conquer Step:** Recursively solve the problem on the subarray $S[2..n]$. Recursion ends when the subarray size is 1; that is, the subarray is $S[n..n]$. In that case, the value of $S[n]$ is set to 1. (This is because the longest ascending sequence starting at $A[n]$ consists of just the single element $A[n]$.)

**Combine Step:** By recursion, we have the correct values for the subarray $S[2..n]$. So, the combine step needs to find the correct value of $S[1]$ using the values in the subarray $S[2..n]$. Note that $A[1]$ starts an ascending subsequence with any element $A[i]$ when $A[1] < A[i]$, $2 \leq i \leq n$. Therefore,

$$S[1] = 1 \text{ if } A[1] > A[i] \text{ for all } i, 2 \leq i \leq n$$
$$= 1 + \max_{2 \leq i \leq n} \{ S[i] : A[1] < A[i] \} \text{ otherwise.}$$

As mentioned earlier, after all the entries of the array $S[1..n]$ are available, the solution to the problem is the largest value in $S$.

Part (b): We will first show the pseudocode for computing the elements of the $S$ array. The general function to be written has form $\text{Compute} (S, A, 1, n)$ where the goal is to compute the entries of the subarray $S[i..n]$. 

---

2
Compute(S, A, i, n) { // Computes the values of the subarray S[i .. n].
    // Uses a local variable temp.
    if (i == n) {
        S[n] = 1;
    }
    else { // Array has two or more elements; must use recursion.
        Compute(S, A, i+1, n); // Divide and conquer steps.
        temp = 0; // Combine step begins here.
        for j = i+1 to n do {
            if ((A[i] < A[j]) and (temp < S[j]))
                temp = S[j];
        }
        S[i] = 1 + temp;
    }
} // End of Compute.

Once we have the Compute function, the main algorithm consists of the following two steps.

1. Compute(S, A, 1, n).
2. Output the largest value in array S.

Part (c): We will establish a recurrence for the running time of Step 1 above and prove that the running time of that step is \( O(n^2) \). Since Step 2 obviously runs in \( O(n) \) time, the running time of the whole algorithm is \( O(n^2) \).

Let \( T(n) \) denote the running time of Step 1 when \( A \) and \( S \) have \( n \) elements each. Examining the pseudocode for the Compute function, we notice the following:

(i) The time for the divide step is \( O(1) \).

(ii) The time for the conquer step is \( T(n - 1) \) since there is only one recursive call which computes the the values of the subarray \( S[2 .. n] \) (of size \( n - 1 \)). When the subarray size is 1, the time is \( O(1) \); that is, \( T(1) = O(1) \).

(iii) The time for the combine step is \( O(n) \) since it spends \( O(1) \) time on each element of the \( n - 1 \) element subarrays \( S[2 .. n] \) and \( A[2 .. n] \).

Therefore, the recurrence for \( T(n) \) is given by

\[
T(n) \leq T(n - 1) + c_1 n \quad \text{for } n \geq 2 \quad \text{and} \quad T(1) = c_2
\]

where \( c_1 \) and \( c_2 \) are constants.

Part (d): We can show that the solution to the above recurrence is \( T(n) = O(n^2) \) using the iteration method as follows.
\[ T(n) \leq T(n-1) + c_1 n \quad \text{(given)} \]
\[ \leq T(n-2) + c_1(n-1) + c_1 n \quad \text{(substituting for } T(n-1)) \]
\[ = T(n-2) + c_1 [(n-1) + n] \quad \text{(simplification)} \]
\[ \leq T(n-3) + c_1 [(n-2) + (n-1) + n] \quad \text{(substituting for } T(n-2)) \]
\[ \vdots \]
\[ \leq T(1) + c_1 [2 + 3 + \ldots + (n-1) + n] = O(n^2). \]

**Problem 3:**
Let \( I_1, I_2, \ldots, I_n \) denote the given items. For item \( I_i \), the weight is \( w_i \) and profit is \( p_i \) \((1 \leq i \leq n)\).

**Details regarding the dynamic programming table:**

The dynamic programming table \( P \) that we use is a two-dimensional array with \( n \) rows (numbered 1 to \( n \)) and \( W + 1 \) columns (numbered 0 through \( W \)). For \( 1 \leq i \leq n \) and \( 0 \leq j \leq W \), entry \( P[i, j] \) stores the maximum profit that can be obtained by choosing a subset of items from \( \{I_1, I_2, \ldots, I_i\} \) with a total weight of at most \( j \).

Once we compute the values of all the entries in \( P \), the solution to the problem is given by \( P[n, W] \).

**Computation of the Table Entries:**

We can initialize Row 1 of the \( P \) matrix using the fact the chosen set can only be either the empty set or the set \( \{I_1\} \). Thus, \( P[1, j] = p_1 \) for \( j \geq w_1 \) and \( P[1, j] = 0 \) otherwise.

Now, assume that for some \( i \geq 2 \), we have computed all the entries of the \( P \) matrix up to Row \( i - 1 \). To compute the entries of Row \( i \), we observe that for the value \( P[i, j] \) \((0 \leq j \leq W)\), there are two possibilities.

(a) There is an optimal solution with weight at most \( j \) for the subset \( \{I_1, \ldots, I_i\} \) that does not include item \( I_i \). In such a case, the value of \( P[i, j] \) is equal to \( P[i-1, j] \).

(b) Every optimal solution with weight at most \( j \) for the subset \( \{I_1, \ldots, I_i\} \) includes item \( I_i \). In such a case, the value of \( P[i, j] \) is equal to \( P[i-1, j-w_i] + p_i \).

Therefore, \( P[i, j] = \max\{P[i-1, j], P[i-1, j-w_i] + p_i\} \), \( 0 \leq j \leq W \).

**Note:** During the computation, if a column index is negative, the corresponding entry of \( P \) is assumed to have the value \(-\infty\) (so that the entry doesn’t play any role in determining the maximum profit).

**Pseudocode for the Algorithm:**

1. Set \( P[1, j] = 0 \) for \( 0 \leq j < w_1 \) and \( P[1, j] = p_1 \) for \( w_1 \leq j \leq W \).

2. for \( i = 2 \) to \( n \) do
   
   for \( j = 0 \) to \( W \) do
     
     \( P[i, j] = \max\{P[i-1, j], P[i-1, j-w_i] + p_i\} \).

3. Output \( P[n, W] \).

**Running Time Analysis:** There are \( O(nW) \) entries in the \( P \) matrix. Using the above algorithm, each entry can be computed in \( O(1) \) time (since it involves finding the larger of just two values). So, the running time of the algorithm is \( O(nW) \).