1a) When i and j are within the range of A's indices and i j, L[i, j] would be the length of the longest palindromic subsequence in A. i and j should satisfy the conditions: a) $1 \le i \le n$, b) $1 \le j \le n$, c) $i \le j$ where n is the length of the sequence. L[1, n] would give the length of the longest palindromic subsequence in the entire sequence A.

1b)

L[i,j] = 1	if i = j
L[i,j] = L[i + 1, j- 1] + 2	if i < j and ai = aj
L[i,j] = max(L[i + 1,j], L[i, j - 1])	if i < j and ai ≠ aj

1c) The dynamic programming algorithm for the Longest Palindromic Subsequence (LPS) problem has a worst-case time complexity of Θ(n^2), where n is the length of the input sequence. This is due to the fact that we must fill a table of size n x n with $1 \le i \le j \le n$ in order to compute L[i, j] for all feasible subsequences. Since computing L[i, j] involves checking conditions, adding, or taking the maximum of two numbers according to the relation we specified before, it requires constant time Θ(1) in the worst case. The total number of operations would be: Since the table contains n(n+1)/2 entries (i.e., half of the table because we only need to populate entries when i ≤ j), Θ(n(n+1)/2 * 1) = Θ(n^2). Therefore, the algorithm's worst-case time complexity is Θ(n^2). The total time complexity of the algorithm would be equal to Θ(n^2) as there are n^2 cells in the table and each cell must be filled in a certain amount of time.

2a) The values of i and j for which P[i, j] represents the maximum value that can be packed into the knapsack of capacity W considering all n items are: i = n j = W Taking into account all n objects (o1, o2,..., on) and the knapsack's capacity of W, P[n, W] reflects the maximum value that may be packed inside.

2b)

P[i, j] = 0	if i = 0
P[i, j] = P[i - 1, j]	if i > 0 and j < wi
max(P[i - 1, j],vi + P[i - 1, j - wi])	if i > 0 and j ≥ wi

2c) Dynamic programming's solution to the 0–1 knapsack issue has an Θ(nW) worst-case time complexity, where n is the number of items and W is the knapsack's capacity. The maximum value that can fit inside a knapsack with capacity j, taking into account the first i items, must be entered into a table with a size of n x W, where each cell (i,j) represents this value. We need to compute P[i,j] based on the recurrence relation in order to populate each cell in the table. Since it entails checking conditions and taking the maximum of two numbers, computing P[i,j] in the worst scenario requires constant time. Since each cell in the table requires a constant amount of time

Θ(1), to fill n x W, the total number of operations would be: Θ(n * W * 1) = Θ(n*W). Hence, the algorithm's total time complexity would be Θ(nW).

3a) A greedy strategy would prioritize objects in the 0-1 Knapsack problem based on their value-to-weight ratio (vi/wi), then decide which object to put in the knapsack at each step.

The greedy choice would be to:
a) Determine the value-to-weight ratio for each object using the formula vi/wi where i = 1, 2,..., n.
b) Sort the objects according to their value-to-weight ratios in decreasing order.
c) Set the knapsack's remaining capacity to W after initializing it as empty.
d) If it fits inside the remaining space in the knapsack, add the object with the highest value-to-weight ratio first. Continue in this manner, taking into account objects with lower value-to-weight ratios, until the knapsack is full or all objects have been taken into consideration.
This greedy strategy might not take into account specific combinations of items that could result in a greater total value, hence it cannot guarantee an optimal solution to the 0-1 Knapsack issue.

3b) The subproblem we are left with concerns the remaining items and the updated capacity of the knapsack after picking object oi by the greedy choice in section (a). If object oi weighs wi and the original knapsack capacity is W, the updated capacity of the knapsack is W - wi after selecting object oi. Finding the most objects that can fit in the knapsack with the updated capacity (W - wi) while taking into account the remaining objects excluding oi, such as (o1, o2,..., oi-1, oi+1,..., on), with weights (w1, w2, ..., wi-1, wi+1, ..., wn) and values (v1, v2, ..., vi-1, vi+1, ..., vn) is the subsequent subproblem. The greedy approach will iteratively solve this subproblem until the knapsack's capacity is exhausted or no more objects can be added.