Problem 1. (25 points)

Let $A = \{a_1, a_2, \ldots, a_n\}$ be a set of $n$ positive integers and let $T$ be another integer. Design a dynamic programming algorithm that determines whether there exists a subset of $A$ whose total sum is exactly $T$. Analyze the time- and space-complexity of your solution.

For instance, if $A = \{4, 5, 17, 23, 11, 2\}$ and $T = 35$ the algorithm should return True because the subset $\{5, 17, 11, 2\}$ sums to 35. For the same set of numbers if we choose $T = 31$ the problem has no solution, and the algorithm will return False.

Answer: This is another version of 01-knapsack called SubsetSum, where there are no benefits and we have to fill the knapsack exactly.

Let $t(i, j) = True$ if there is a subset of the first $i$ items that has a total of exactly $j$, false otherwise. We have

$$
t(i, j) = \begin{cases} 
(j = 0) & \text{if } i = 0 \\
t(i-1, j) & \text{if } i > 0 \text{ AND } a_i > j \\
t(i-1, j) \text{ OR } t(i-1, j-a_i) & \text{if } i > 0 \text{ AND } a_i \leq j 
\end{cases}
$$

We want to know $t(n, T)$. Time complexity is $O(nT)$. Space complexity is $O(n + T)$.

Problem 2.

You have a set of $n$ jobs to process on a machine. Each job $j$ has a processing time $t_j$, a profit $p_j$ and a deadline $d_j$. The machine can process only one job at a time, and job $j$ must run uninterrupted for $t_j$ consecutive units of time. If job $j$ is completed by its deadline $d_j$, you receive a profit $p_j$, otherwise a profit of 0. You can assume that all parameters are integers, and that the jobs are sorted in increasing order of deadline. Give a dynamic programming algorithm to the problem of determining the schedule that gives the maximum amount of profit. Analyze the time- and space-complexity of your solution.

Answer: Define $P[i]$ to be the max profit that can be obtained by scheduling jobs with deadlines less or equal to $d_i$ and with the last job being $i$. Then

$$
P[i] = \begin{cases} 
0 & \text{if } i = 0 \\
p_i + \max_{0 \leq k < i} \{P[k] : d_k + t_i \leq d_i\} & \text{otherwise} 
\end{cases}
$$

Time complexity is $O(n^2)$ because it takes linear time to fill each entry of the array. Space complexity is $O(n)$.

Problem 3.

Let $A$ be a $n \times m$ matrix of 0’s and 1’s. Design a dynamic programming $O(nm)$ time algorithm for finding the largest square block of $A$ that contains 1’s only.

Hint: Define the dynamic programming table $l(i, j)$ be the length of the side of the largest square block of 1’s whose bottom right corner is $A[i, j]$.

Answer: Let $l(i, j)$ be the length of the side of the largest square block of 1’s whose bottom right corner is $A[i, j]$. Then:

$$
l(i, j) = \begin{cases} 
0 & \text{if } A[i, j] = 0 \\
\min\{l(i-1, j-1), l(i-1, j), l(i, j-1)\} + 1 & \text{otherwise} 
\end{cases}
$$
Once the matrix \( l \) is computed, simply scan for the largest number. Time complexity is \( O(nm) \), space complexity is \( O(n + m) \).

**Problem 4.**

A string \( y \) is a *palindrome* if \( y^R = y \), where \( y^R \) is the reverse of \( y \). Given a text \( x \) a partitioning of \( x \) is a *palindrome partitioning* if every substring of the partition is a palindrome. For example, \( \text{aba|bb|a|bb|a|b|aba} \) and \( \text{aba|bbabb|ababa} \) are two palindrome partitioning of \( x = \text{ababbbabababa} \). Design a dynamic programming algorithm to determine the coarsest (i.e., fewest cuts) palindrome partitioning of \( x \). In the example, the second partition (3 cuts) is optimal. Remember to analyze the time- and space-complexity of your solution.

**Answer:** We need two tables for this problem. Define \( P[i, j] = \text{True} \) if \( a_i a_{i+1} \ldots a_j \) and is a palindrome, for all \( 1 \leq i \leq j \leq n \). Let us also define \( C[i] = \text{number of cuts in the best palindrome partition } a_1 a_2 \ldots a_i \). We have

\[
P[i, j] = \begin{cases} 
a_i = a_j \text{ AND } P[i + 1, j - 1] & \text{if } i < j - 1 \\
a_i = a_j & \text{if } i = j - 1 \\
\text{True} & \text{if } i = j
\end{cases}
\]

\[
C[i] = \begin{cases} 
\min_{k \in [0, i-1]} \{ C[k] + 1 : P[k + 1, i] = \text{True} \} & \text{if } i > 0 \\
0 & \text{if } i = 0
\end{cases}
\]

The algorithm first computes \( P[i, j] \) and then \( C[i] \). The complexity is \( O(n^2) \).