1 Knapsack Problem Analysis and Solution

1.1 Problem Definition

The Knapsack Problem can be defined as follows: Given a set of n items, where each item i has:

- Profit $p_i \in \mathbb{R}^+$ (a non-negative real number),
- Weight $w_i \in \mathbb{R}^+$ (a non-negative real number),

and a knapsack with a total capacity $W \in \mathbb{R}^+$, determine the subset of items to include in the knapsack to maximize the total profit without exceeding the weight constraint.

The **0/1 Knapsack Problem** is known to be NP-complete, which means that no known polynomial-time algorithm can solve all instances of the problem efficiently. This has significant implications for practical applications, where approximation or heuristic methods are often employed when dealing with large problem sizes. On the other hand, the **Fractional Knapsack Problem** can be solved optimally in polynomial time using a greedy approach, making it a much simpler variant computationally.

1.2 Mathematical Generalization

Let:

$$\mathbf{p} = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{bmatrix}, \quad \mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}, \quad \text{represent the profit and weight vectors, respectively, in } \mathbb{R}^n.$$

Define a binary decision vector:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad x_i \in \{0, 1\},$$

where $x_i = 1$ if item i is included in the knapsack, and $x_i = 0$ otherwise.

The objective is to maximize the total profit:

$$\max \mathbf{p}^\top \mathbf{x} = \sum_{i=1}^{n} p_i x_i,$$

subject to the weight constraint:

$$\mathbf{w}^\top \mathbf{x} = \sum_{i=1}^{n} w_i x_i \leq W,$$

and the binary decision constraint:

$$x_i \in \{0,1\}, \quad \forall i = 1, \ldots, n.$$

1.3 Problem Structure

Optimal Substructure

The Knapsack Problem exhibits **optimal substructure**, meaning the solution to a problem can be constructed from the solutions of its subproblems. Specifically, for n items and a capacity W, the maximum profit $V(n, W)$ can be determined as:

$$V(n,W) = \begin{cases} 0, & \text{if } n = 0 \text{ or } W = 0, \\ V(n-1, W), & \text{if } w_n > W, \\ \max\{V(n-1,W), p_n + V(n-1, W - w_n)\}, & \text{otherwise.} \end{cases}$$

This means that if the weight of the n-th item exceeds W, it cannot be included, and the problem reduces to $V(n-1, W)$. Otherwise, we choose the maximum between excluding the n-th item and including it.

Key Intuition:

Dynamic programming allows us to systematically explore all possible subsets of items while avoiding redundant calculations. The realization comes from analyzing the problem as a sequence of decisions for each item:

- **Include the item:** The item's profit is added, and the capacity decreases by the item's weight.

- **Exclude the item:** The problem reduces to the previous state with the same capacity.

These choices are naturally represented as states in a table, where each cell corresponds to the maximum profit attainable for a given capacity and set of items.

The recursive relation:

$$dp[i][w] = \begin{cases} dp[i-1][w], & \text{if } w_i > w, \\ \max(dp[i-1][w], p_i + dp[i-1][w-w_i]), & \text{otherwise,} \end{cases}$$

captures this logic, enabling us to solve the problem iteratively.

1.4 Expanding the Dynamic Programming Solution

Constructing the Table:

The table $dp[i][w]$ stores the maximum profit for the first i items and a knapsack of capacity w. The table is filled row by row:

1. If the current item's weight w_i exceeds the current capacity w, the item cannot be included, so:

$$dp[i][w] = dp[i-1][w].$$

2. Otherwise, we choose the better option:

$$dp[i][w] = \max(dp[i-1][w], p_i + dp[i-1][w-w_i]).$$

Result Extraction:

The solution is found in $dp[n][W]$, where n is the number of items and W is the total capacity.

Traceback (Optional):

To reconstruct the selected items:

- Start from $dp[n][W]$.
- If $dp[i][w] \neq dp[i-1][w]$, item i was included, and $w \leftarrow w - w_i$.
- Continue until $i = 0$.

This process ensures the selection of the optimal subset.

2 Minimum Coin Change Problem

2.1 Problem Statement

Given a set of n coin denominations $\{v_1, v_2, \ldots, v_n\}$ where $v_1 = 1$, determine the minimum number of coins required to form any amount $A \geq 0$.

2.2 Recursive Solution Formulation

Let $C(k)$ represent the minimum number of coins needed to make an amount k. The recursive relationship is defined as follows:

$$C(k) = \begin{cases} 0, & \text{if } k = 0, \\ \infty, & \text{if } k < 0, \\ \min_{i:v_i \leq k}\{1 + C(k - v_i)\}, & \text{if } k > 0. \end{cases}$$

The base cases handle the scenarios where no coins are needed ($k = 0$) or where the amount cannot be formed ($k < 0$). For $k > 0$, the solution is derived by considering all valid coin denominations $v_i \leq k$ and choosing the one that minimizes the total number of coins.

2.3 Key Observations: Optimal Substructure

The Minimum Coin Change problem exhibits *optimal substructure*, which means that the optimal solution to a problem can be constructed from the optimal solutions to its subproblems. In this case, the problem of finding the minimum number of coins required to make a target sum k can be solved by considering the optimal solutions to smaller subproblems. Specifically, the optimal number of coins needed for a sum k depends on the optimal solutions to the subproblems of sums $k - v_i$, where v_i is a coin denomination.

2.3.1 Mathematical Justification of Optimal Substructure

The recurrence relation demonstrates the optimal substructure property of the problem. For a given sum k, the solution depends on the optimal solutions to the subproblems for all sums $k - v_i$ (where v_i is a coin denomination). By considering each possible coin denomination, the algorithm finds the optimal number of coins needed for k by selecting the smallest possible number of coins from all possible subproblems. Thus, the solution for k is built from solutions to smaller subproblems.

$$C(k) = \min_{i:v_i \leq k} (1 + C(k - v_i))$$

This recurrence clearly illustrates that the minimum number of coins for a target sum k depends on the minimum number of coins for each smaller target sum $k - v_i$. As a result, the problem is broken into smaller subproblems, each of which is solved independently and contributes to the solution of the larger problem.

2.3.2 Proof of Optimal Substructure

To prove that the problem has optimal substructure, we can observe that the solution to the problem involves recursively solving smaller subproblems. Each decision made at a higher level of the recursion tree (choosing a coin v_i) is based on the optimal solution to the subproblem of making the amount $k - v_i$. Hence, the solution to the problem is constructed from the optimal solutions to smaller subproblems.

In the case of dynamic programming, this optimal substructure is leveraged by solving each subproblem once and storing the results to avoid redundant work. The value $C(k)$ is built from smaller subproblems, and each subproblem is solved optimally to contribute to the global solution.

2.4 Subproblem Overlap and Dynamic Programming

"The second key property that supports optimal substructure is the presence of *overlapping subproblems*, where the same subproblems are solved multiple times during the computation.". For example, when calculating $C(k)$, the subproblem for $C(k - v_i)$ is computed multiple times for different values of k and v_i. This redundancy can be avoided by storing the results of subproblems, as done in dynamic programming approaches such as memoization or bottom-up tabulation.

2.4.1 Memoization

Memoization, also known as top-down dynamic programming, utilizes recursion while storing the results of each subproblem. This process ensures that if a subproblem is encountered again, its result can be retrieved from the memoization table, instead of recomputing it. The recursive structure of the problem allows for the computation of each subproblem only once. The time complexity of the memoized approach can be derived as follows:

$$T_{\text{memo}}(n, k) = O(n \cdot k)$$

Here, n is the number of coin denominations, and k is the target sum. The function will compute each subproblem once, where each subproblem is defined by a specific combination of coin denominations and target sum. Thus, the total number of subproblems is $O(n \cdot k)$, and each subproblem takes constant time to solve, leading to an overall time complexity of $O(n \cdot k)$.

2.4.2 Tabulation

The tabulation approach, also known as bottom-up dynamic programming, constructs the solution iteratively, starting from smaller subproblems and gradually building up to the desired result. In this approach, a table is used to store the results of the subproblems, ensuring that each subproblem is solved once. The time complexity for tabulation is also $O(n \cdot k)$, as we fill a table of size $n \times k$ in a bottom-up manner, iterating through all possible subproblems.

$$T_{\text{tab}}(n, k) = O(n \cdot k)$$

Thus, the time complexity for both memoization and tabulation approaches is linear with respect to the number of subproblems and the number of coin denominations.

2.5 Space Complexity Analysis

Memoization Space Complexity In the memoization approach, space is required to store the results of subproblems in a memoization table. The table contains an entry for each combination of coin denomination and target sum, which results in a space complexity of $O(n \cdot k)$. Additionally, there is space needed for the recursive stack, which in the worst case can grow to a depth of $O(k)$. Therefore, the overall space complexity for memoization is:

$$S_{\text{memo}}(n, k) = O(n \cdot k + k)$$

For large values of k, the term $O(n \cdot k)$ dominates.

Tabulation Space Complexity In the tabulation approach, space is used only for the table that stores the results of the subproblems. This table has a size of $O(n \cdot k)$, and there is no recursion stack. Therefore, the space complexity for tabulation is:

$$S_{\text{tab}}(n, k) = O(n \cdot k)$$

Resources:

- Code – Minimum Coins to Make Sum
- Intuition – Minimum Coins to Make Sum

3 Finding the Maximum Number of Non-Intersecting Line Segments Using DP

In this section, we tackle a problem involving two sets of points on two parallel lines. The points on the first line have coordinates $(a_1, 1), (a_2, 1), \ldots, (a_n, 1)$, and the points on the second line have coordinates $(b_1, -1), (b_2, -1), \ldots, (b_n, -1)$. The task is to find the maximum number of non-intersecting line segments that can be drawn between these points, where each segment connects a point from the first line to a point on the second line, and the segments do not overlap.

3.1 Problem Definition

Given two sets of points:

- $P_i = (a_i, 1)$ for $i = 1, \ldots, n$ on the line $y = 1$, with $a_1, a_2, \ldots, a_n$ being the x-coordinates of the points.
- $Q_i = (b_i, -1)$ for $i = 1, \ldots, n$ on the line $y = -1$, with $b_1, b_2, \ldots, b_n$ being the x-coordinates of the points.

The points are indexed from 1 to n, but the indexing does not necessarily correspond to the order of the x-coordinates. The goal is to draw as many non-intersecting line segments as possible, where each segment connects a point P_i from the first line to a point Q_j from the second line. Two segments do not intersect if and only if their x-coordinates are ordered similarly on both lines: If a segment connects P_i to Q_j and another segment connects P_k to Q_l, the segments do not intersect if $i < k$ and $j < l$, or $i > k$ and $j > l$.

The task is to find the maximum number of such non-intersecting segments.

3.2 Dynamic Programming Approach

To solve the problem, we use a dynamic programming approach, which leverages the structural properties of optimal substructure and overlapping subproblems. These properties are analyzed below.

3.2.1 Optimal Substructure

The condition for connecting the i-th point on $y = 1$ to the j-th point on $y = -1$ is that the connection does not create intersections with previously drawn segments. This is equivalent to ensuring that the ordering of the indices i and j respects the order in which points are connected.

This recurrence relation ensures that:

- If a line segment can be drawn between the i-th point on $y = 1$ and the j-th point on $y = -1$ without intersecting previously established segments, then the problem reduces to solving for the first $i - 1$ and $j - 1$ points and adding 1.
- Otherwise, we must maximize the solution by either excluding the i-th point on $y = 1$ or the j-th point on $y = -1$.

Specifically, if we denote by $M(i, j)$ the maximum number of non-intersecting line segments that can be drawn between the first i points on the line $y = 1$ and the first j points on the line $y = -1$, the following recurrence relation holds:

$$M(i,j) = \begin{cases} M(i-1, j-1) + 1, & \neg\text{intersection} \\ \max(M(i-1, j), M(i, j-1)), & \text{otherwise.} \end{cases}$$

The problem's optimal solution uses the sub-problems' optimal solutions so it exhibits optimal substructure.

3.3 Equivalence Between Maximum Disjoint Segments problem and LCS

Each point a_i on $y = 1$ has an associated index $I(a_i)$, and each point b_j on $y = -1$ has an associated index $I(b_j)$, corresponding to their relative positions in their respective lines. The task now translates to matching indices $I(a_i) = I(b_j)$ with straight line segments while ensuring that no two segments intersect.

In the context of the given geometric problem, we have:

1. n points on the line $y = 1$ with x-coordinates $\{a_1, a_2, \ldots, a_n\}$,
2. n points on the line $y = -1$ with x-coordinates $\{b_1, b_2, \ldots, b_n\}$.

The equivalence between the problems of finding the maximum number of non-intersecting line segments and computing the Longest Common Subsequence (LCS) can be understood intuitively by considering the constraints of each problem.

3.3.1 Geometric Constraints and Order Preservation

To avoid intersections, the connections between points must preserve the order of indices:

- If $I(a_i) = I(b_j)$ is connected, and $I(a_{i'}) = I(b_{j'})$ is also connected, then the order of connections must satisfy:

$$i < i' \quad \text{and} \quad j < j'.$$

  This means that both the sequence of indices on $y = 1$ (i) and $y = -1$ (j) must be strictly increasing.

This order-preserving property is precisely the defining characteristic of a subsequence in the Longest Common Subsequence (LCS) problem. A valid LCS aligns elements from two sequences while maintaining their relative order, which corresponds exactly to non-intersecting connections in the geometric problem.

3.3.2 Mapping to the LCS Problem

Given the sequences:

$$\{I(a_1), I(a_2), \ldots, I(a_n)\} \quad \text{and} \quad \{I(b_1), I(b_2), \ldots, I(b_n)\},$$

we observe the following:

1. Each pair $(I(a_i), I(b_j))$ in the LCS corresponds to a valid, non-intersecting line segment in the geometric problem.
2. The maximum number of non-intersecting line segments is thus equivalent to the length of the LCS of these two sequences.

By reducing the geometric problem to the LCS problem, we leverage the well-known algorithms for computing the LCS to solve the problem of maximum disjoint line segments efficiently. In the following sections, we formalize this equivalence through rigorous proof.

3.3.3 Proof of Theorem

We aim to prove that the maximum number of disjoint line segments that can be drawn between points on the lines $y = 1$ and $y = -1$ is equal to the length of the Longest Common Subsequence (LCS) of the sequences:

$$\{I(a_1), I(a_2), \ldots, I(a_n)\} \quad \text{and} \quad \{I(b_1), I(b_2), \ldots, I(b_n)\},$$

where $I(a_i)$ is the index of the i-th point on $y = 1$ (ordered by their x-coordinates), and $I(b_j)$ is the index of the j-th point on $y = -1$ (also ordered by their x-coordinates). To do so, we prove the following two inequalities:

1. **Number of Disjoint Segments $\geq$ Length of LCS:** Any valid Longest Common Subsequence alignment corresponds to a set of non-intersecting line segments.
2. **Number of Disjoint Segments $\leq$ Length of LCS:** Any set of non-intersecting line segments corresponds to a subsequence alignment, which cannot be longer than the LCS.

—

3.3.4 Part 1: Number of Disjoint Segments $\geq$ Length of LCS

Let us consider the LCS of the sequences $\{I(a_1), I(a_2), \ldots, I(a_n)\}$ and $\{I(b_1), I(b_2), \ldots, I(b_n)\}$. Each match in the LCS corresponds to a pair of points (i, j) where:

- $I(a_i) = I(b_j)$, meaning that the points on $y = 1$ and $y = -1$ have matching indices, and
- The order of indices is preserved, i.e., if $i < i'$ and $j < j'$, then (i, j) and (i', j') are valid pairs without crossing.

Since the LCS ensures that the order of indices is preserved, no two line segments corresponding to the LCS matches will intersect. Thus, the LCS alignment gives a valid set of non-intersecting line segments.

Conclusion: The length of the LCS corresponds to a valid number of non-intersecting line segments, so the number of disjoint segments is at least the length of the LCS.

—

3.3.5 Part 2: Number of Disjoint Segments $\leq$ Length of LCS

Now, consider the maximum number of disjoint line segments that can be drawn. Let this number be k, corresponding to k valid pairs $(i_1, j_1), (i_2, j_2), \ldots, (i_k, j_k)$. For the line segments to be non-intersecting:

- The indices $i_1, i_2, \ldots, i_k$ on $y = 1$ must be strictly increasing, and
- The indices $j_1, j_2, \ldots, j_k$ on $y = -1$ must also be strictly increasing.

These conditions ensure that the sequence of matched indices forms a subsequence of both $\{I(a_1), I(a_2), \ldots, I(a_n)\}$ and $\{I(b_1), I(b_2), \ldots, I(b_n)\}$. By definition, the length of any such subsequence cannot exceed the length of the LCS.

Conclusion: The number of disjoint line segments is bounded above by the length of the LCS.

—

3.3.6 Combining the Two Parts

From Part 1, we know that the number of disjoint line segments is at least the length of the LCS. From Part 2, we know that the number of disjoint line segments is at most the length of the LCS. Combining these results, we conclude that:

$$\text{Maximum Number of Disjoint Segments} = \text{Length of LCS}.$$

—

3.4 Algorithm Simplification

Since the problem is equivalent to finding the LCS of the two sequences, we can directly solve it using the standard dynamic programming approach for LCS. The LCS table M is computed as follows:

$$M(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ M(i-1,j-1)+1 & \text{if } I(a_i) = I(b_j), \\ \max(M(i-1,j), M(i,j-1)) & \text{otherwise.} \end{cases}$$

This has a time complexity of $O(n^2)$ and a space complexity of $O(n^2)$, where n is the number of points.

Acknowledgements

This work was completed as part of the course *Algorithm Design and Analysis* offered by the Department of Applied Mathematics at the University of Crete.
Course Code: TEM3062