CS 2500: Algorithms

Lecture 26: Dynamic Programming: Unbounded Knapsack Problem

Shubham Chatterjee

Missouri University of Science and Technology, Department of Computer Science

November 14, 2024

Unbounded Knapsack Problem

We are given a number of objects and a knapsack.

- Unlike the 0/1 Knapsack Problem, each object **can be taken multiple times**.
- Let $i = 1, 2, \ldots, n$ denote the objects.
- Each object i has:
  - a positive weight w_i
  - a positive value v_i
- The knapsack has a weight capacity W.
- Goal: Fill the knapsack in a way that maximizes the **total value of the included objects**.

Let x_i represent the number of times object i is included in the knapsack, where $x_i \geq 0$ and is an integer.

Unbounded Knapsack Problem

Mathematical Formulation

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{n} x_i v_i \\ \text{subject to} \quad & \sum_{i=1}^{n} x_i w_i \leq W \end{aligned}$$

where:

- $v_i > 0$ and $w_i > 0$
- $x_i \in \mathbb{Z}_{\geq 0}$ for $1 \leq i \leq n$

Constraints:

- v_i and w_i are constants for each item
- x_i are variables in the solution.

Objective: Maximize the total value without exceeding the weight capacity W.

Difference: 0/1 Knapsack and Unbounded Knapsack

- **Choice of Items:**
  - **0/1 Knapsack:** Each item can be included at most once in the knapsack.
  - **Unbounded Knapsack:** Each item can be included multiple times, allowing for unlimited instances of each item.
- **Problem Constraints:**
  - **0/1 Knapsack:** Binary decision for each item — either take it or leave it (0 or 1).
  - **Unbounded Knapsack:** For each item, there is flexibility to take as many instances as needed, as long as the total weight does not exceed the knapsack's capacity.

Difference: 0/1 Knapsack and Unbounded Knapsack

- **Optimal Substructure:**
  - **0/1 Knapsack:** Requires a decision for each item, making it suitable for a **2D DP table** (items by capacity).
  - **Unbounded Knapsack:** Focuses on each weight limit independently, making it suitable for a **1D DP table** with each weight capacity considered separately.
- **Typical Applications:**
  - **0/1 Knapsack:** Used when items are indivisible and can be selected only once (e.g., selecting projects within budget).
  - **Unbounded Knapsack:** Used when items can be reused multiple times (e.g., coin change problems, resource allocation).

Key Idea for Unbounded Knapsack

In the Unbounded Knapsack problem:

- When we include an item i with weight w_i and value v_i, we reduce the remaining capacity by w_i.
- After including it once, we still have the option to include it again, as long as the remaining capacity $w - w_i$ is sufficient.

Why Only w Is Needed in Unbounded Knapsack

This is a fundamental difference between the 0/1 Knapsack and the Unbounded Knapsack problem.

- In the 0/1 Knapsack problem, each item can be chosen only once. Therefore, we track both the item index i and the remaining weight w to make sure we don't revisit the same item.
- This is why the recursive function for 0/1 Knapsack often looks like Knapsack(i, w), where i tracks which items have already been considered.

Why Only w Is Needed in Unbounded Knapsack

- In the Unbounded Knapsack problem, each item can be chosen multiple times.
- This makes the item index i less relevant in the recursion, as we are allowed to reuse any item as many times as we want, as long as it fits within the remaining weight.
- Since items can be reused, we do not need a separate i index to control which items can be included. We simply need to know:
  1. The remaining weight w.
  2. For each weight w, we evaluate all items i independently, as they can all be reconsidered multiple times.
- Thus, the recursion only depends on w, and for each value of w, we check each item i to see if it can fit.

Recurrence Relation for Unbounded Knapsack

$$\text{Knapsack}(w) = \max\left(\text{Knapsack}(w),\ \text{Knapsack}(w - w_i) + v_i\right) \quad \text{for each } i \text{ where } w_i \leq w$$

How the Recurrence Handles Multiple Inclusions

1. If we include item i:
   - We compute Knapsack$(w - w_i) + v_i$, which gives us the maximum value achievable with the reduced capacity $w - w_i$ plus the value of item i.
   - This means that after including item i, we are left with a subproblem of capacity $w - w_i$.
2. The subproblem Knapsack$(w - w_i)$ is solved independently:
   - Because Knapsack(w) is defined as the maximum value achievable with capacity w, if i fits within $w - w_i$, it can be chosen again in the solution for Knapsack$(w - w_i)$.
   - Therefore, the recurrence will naturally allow for item i to be included multiple times because every time we consider Knapsack$(w - w_i)$, item i is again a candidate for inclusion.

Example to Illustrate Multiple Inclusions

Suppose we have an item with:

- Weight $w_1 = 2$
- Value $v_1 = 3$

And we want to maximize the value for a knapsack with capacity $W = 6$.

Using the recurrence:

1. First, we calculate Knapsack(6).
2. We check if we can include item 1 (since $w_1 = 2 \leq 6$):
   - If we include item 1, we get $v_1 + \text{Knapsack}(6-2) = 3 + \text{Knapsack}(4)$.
3. For Knapsack(4):
   - Again, we can include item 1, resulting in $3 + \text{Knapsack}(2)$.
4. For Knapsack(2):
   - We can include item 1 one more time, giving $3 + \text{Knapsack}(0) = 3$ (since $\text{Knapsack}(0) = 0$).

The total value achieved by including item 1 multiple times is $3 + 3 + 3 = 9$, which correctly accounts for multiple inclusions.

Unbounded Knapsack: Top-Down Approach with Memoization

Steps for Top-Down Approach:

1. Start with the original problem (e.g., max value with full weight capacity and all items available).
2. Recursively explore each option:
   - If item i is not included, recursively compute the solution without item i.
   - If item i is included, recursively compute the solution with reduced capacity $W - w_i$ **but with** i **still available** (as it can be chosen multiple times).
3. Store results in a table (memoization) as each subproblem is solved.
4. Retrieve results from the table when the same subproblem is encountered again, avoiding redundant computation.

Unbounded Knapsack: Top-Down Approach with Memoization

Algorithm 1 Unbounded Knapsack: Top-Down Approach with Memoization

Require: W: maximum weight capacity of the knapsack
Require: n: number of items
Require: w: array of item weights, v: array of item values
Ensure: Maximum achievable value with capacity W

Initialize a memoization array memo$[0 \dots W]$ with values -1
function KNAPSACK(W)
  if $W == 0$ **then**
    return 0 ▷ Base case: no capacity left
  end if
  if memo$[W] \neq -1$ **then**
    return memo$[W]$ ▷ Return already computed result
  end if
  max_value $\leftarrow 0$
  for $i = 1$ to n **do**
    if $w[i] \leq W$ **then**
      max_value $\leftarrow \max(\text{max\_value}, v[i] + \text{Knapsack}(W - w[i]))$
    end if
  end for
  memo$[W] \leftarrow$ max_value
  return memo$[W]$
end function
return KNAPSACK(W) ▷ Compute the maximum value for full capacity

Unbounded Knapsack: Bottom-Up Approach

Steps for Bottom-Up Approach:

1. Initialize a list (e.g., 1D array) where each entry represents a subproblem (e.g., max value achievable with a specific weight capacity).
2. Fill the list iteratively, starting from the smallest subproblems (e.g., capacity 0).
3. For each item i and each weight w, compute the maximum value achievable by including i multiple times if possible.
4. The final entry in the list gives the answer to the original problem.

Unbounded Knapsack: Bottom-Up Approach

- Define a list $V[w]$ where:
  - w: The weight limit (from 0 to W).
  - $V[w]$: The maximum value achievable with weight capacity w.
- Recurrence relation:

$$V[w] = \max_{i=1}^{n} \begin{cases} V[w - w_i] + v_i & \text{if } w_i \leq w \\ V[w] & \text{otherwise} \end{cases}$$

- Boundary conditions:
  - $V[0] = 0$ (zero weight capacity).

Unbounded Knapsack: Bottom-Up Approach

Algorithm 2 Unbounded Knapsack: Bottom-Up Approach

Require: W: maximum weight capacity of the knapsack
Require: n: number of items
Require: w: array of item weights, v: array of item values
Ensure: Maximum achievable value with capacity W

Initialize a DP array $\mathrm{dp}[0 \ldots W]$ with values 0
for $i = 1$ to W **do**
    for $j = 1$ to n **do**
        if $w[j] \leq i$ **then**
            $\mathrm{dp}[i] \leftarrow \max(\mathrm{dp}[i], v[j] + \mathrm{dp}[i - w[j]])$
        end if
    end for
end for
return $\mathrm{dp}[W]$ $\triangleright$ Maximum value achievable with full capacity W

Unbounded Knapsack Example: Problem

Suppose we have:

- Maximum weight capacity $W = 5$
- Number of items $n = 3$
- Item weights $w = [1, 3, 4]$
- Item values $v = [10, 40, 50]$

Our goal is to maximize the total value without exceeding the weight capacity W.

Unbounded Knapsack Example: Initialization

Initialize the DP array $dp[0 \ldots W]$ with values 0:

$$dp = [0, 0, 0, 0, 0, 0]$$

Unbounded Knapsack Example: Capacity $i = 1$

For capacity $i = 1$:

- Item 1 ($w[1] = 1$, $v[1] = 10$):

  $$dp[1] = \max(dp[1], v[1]+dp[1-w[1]]) = \max(0, 10+dp[0]) = 10$$

- Items 2 and 3: Skipped as their weights exceed $i = 1$.

Updated dp:

$$dp = [0, 10, 0, 0, 0, 0]$$

Unbounded Knapsack Example: Capacity $i = 2$

For capacity $i = 2$:

- Item 1 ($w[1] = 1$, $v[1] = 10$):

  $$dp[2] = \max(dp[2], v[1] + dp[2 - w[1]]) = \max(0, 10 + dp[1]) = 20$$

- Items 2 and 3: Skipped as their weights exceed $i = 2$.

Updated dp:

$$dp = [0, 10, 20, 0, 0, 0]$$

Unbounded Knapsack Example: Capacity $i = 3$

For capacity $i = 3$:

- Item 1 ($w[1] = 1$, $v[1] = 10$):

  $$dp[3] = \max(dp[3], v[1]+dp[3-w[1]]) = \max(0, 10+dp[2]) = 30$$

- Item 2 ($w[2] = 3$, $v[2] = 40$):

  $$dp[3] = \max(dp[3], v[2]+dp[3-w[2]]) = \max(30, 40+dp[0]) = 40$$

- Item 3: Skipped as its weight exceeds $i = 3$.

Updated dp:

$$dp = [0, 10, 20, 40, 0, 0]$$

Unbounded Knapsack Example: Capacity $i = 4$

For capacity $i = 4$:

- Item 1 ($w[1] = 1$, $v[1] = 10$):

  $$dp[4] = \max(dp[4], v[1]+dp[4-w[1]]) = \max(0, 10+dp[3]) = 50$$

- Item 2 ($w[2] = 3$, $v[2] = 40$):

  $$dp[4] = \max(dp[4], v[2]+dp[4-w[2]]) = \max(50, 40+dp[1]) = 50$$

- Item 3 ($w[3] = 4$, $v[3] = 50$):

  $$dp[4] = \max(dp[4], v[3]+dp[4-w[3]]) = \max(50, 50+dp[0]) = 50$$

Updated dp:

$$dp = [0, 10, 20, 40, 50, 0]$$

Unbounded Knapsack Example: Capacity $i = 5$

For capacity $i = 5$:

- Item 1 ($w[1] = 1,\ v[1] = 10$):

  $$dp[5] = \max(dp[5], v[1]+dp[5-w[1]]) = \max(0, 10+dp[4]) = 60$$

- Item 2 ($w[2] = 3,\ v[2] = 40$):

  $$dp[5] = \max(dp[5], v[2]+dp[5-w[2]]) = \max(60, 40+dp[2]) = 60$$

- Item 3 ($w[3] = 4,\ v[3] = 50$):

  $$dp[5] = \max(dp[5], v[3]+dp[5-w[3]]) = \max(60, 50+dp[1]) = 60$$

Final dp:

$$dp = [0, 10, 20, 40, 50, 60]$$

Unbounded Knapsack Example: Final Result

After processing all capacities, the maximum achievable value with capacity $W = 5$ is stored in $dp[5]$:

$$dp[5] = 60$$

Therefore, the maximum achievable value with a knapsack capacity of 5 is $\mathbf{60}$.

Rod Cutting Problem

- A company wants to cut rods into pieces to maximize revenue.
- Each piece cut from a rod has a length i and price p_i.
- Objective: Determine the maximum revenue obtainable by cutting a rod of length n inches, given a table of prices p_i for each rod length i.

Rod Cutting Problem: Formal Problem Definition

- Given:
  - A rod of length n and a price table $p_1, p_2, \ldots, p_n$.
- Goal:
  - Maximize revenue r_n by deciding where to cut the rod.
  - Formula: $r_n = \max\{p_n, p_1 + r_{n-1}, p_2 + r_{n-2}, \ldots, p_{n-1} + r_1\}$.
- Optimal substructure property: Each optimal solution to a problem incorporates optimal solutions to related subproblems.

Rod Cutting Problem: Recurrence Relation

Defining the Subproblem

- Let r_n represent the maximum revenue obtainable for a rod of length n.
- For each length $k \leq n$, r_k represents the maximum revenue for a rod of length k.
- The problem can be broken down into smaller subproblems by considering cuts at different lengths.

Considering All Possible Cuts

- To maximize revenue for a rod of length n, we need to consider all possible lengths for the first cut.
- If the first cut is at length i:
  - We get a piece of length i (revenue $= p_i$).
  - The remaining length is $n - i$, with maximum revenue r_{n-i}.
- For each cut i, the total revenue is $p_i + r_{n-i}$.

Rod Cutting Problem: Recurrence Relation

Base Case

- If there is no rod (length $n = 0$), the revenue is zero:

$$r_0 = 0$$

Rod Cutting Problem: Recurrence Relation

Recursive Case

- To find the maximum revenue r_n for a rod of length n, we consider all possible cuts:

$$r_n = \max_{1 \leq i \leq n} \{p_i + r_{n-i}\}$$

- This formula captures the idea that the revenue for a rod of length n can be obtained by maximizing over all possible ways of making a first cut.

Rod Cutting Problem: Recurrence Relation

Recurrence Relation

$$r_n = \begin{cases} 0 & \text{if } n = 0, \\ \max_{1 \le i \le n}\{p_i + r_{n-i}\} & \text{if } n > 0. \end{cases}$$

- r_n: Maximum revenue for a rod of length n.
- p_i: Price of a piece of rod of length i.
- r_{n-i}: Maximum revenue for the remaining rod of length $n - i$.

Rod Cutting Problem: Recurrence Relation

Key Points

- The recurrence relation divides the problem into subproblems, where each subproblem is a smaller rod-cutting problem.
- The recurrence leverages the **optimal substructure** of the problem: each solution to a rod of length n can be built from solutions of shorter lengths.
- This recurrence relation serves as the basis for both top-down (memoized) and bottom-up dynamic programming solutions.

Rod Cutting Problem: Example

Problem: Sterling Enterprises is a company that buys long steel rods and cuts them into shorter segments to maximize revenue. The company has a price table (as shown below) which indicates the selling price for rods of different lengths. The objective is to determine the optimal way to cut a rod of length n inches to maximize the revenue generated by selling the pieces.

For $n = 4$, consider possible ways to cut the rod into pieces of length i where $1 \leq i \leq n$, and calculate the revenue for each cut.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Rod Cutting Problem: Example

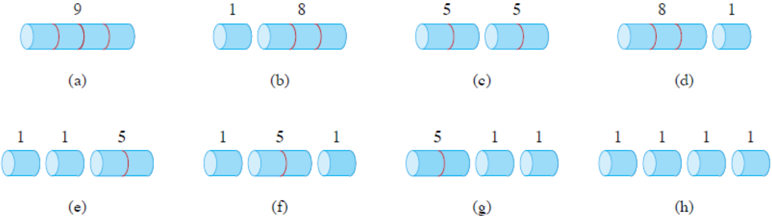

Figure: The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece, according to the sample price table on Slide 27. The optimal strategy is part (c)–cutting the rod into two pieces of length 2–which has total value 10.

Rod Cutting Problem: Top-Down with Memoization

Algorithm 1 Rod Cutting - Top-Down with Memoization

Require: p: Array of prices for each length, n: Length of the rod
Ensure: Maximum revenue obtainable for a rod of length n

function $\textsc{CutRod}(p,\ n)$
  for $i = 0$ to n **do**
    $r[i] \leftarrow -\infty$
  end for
  return $\textsc{CutRodAux}(p,\ n,\ r)$
end function
function $\textsc{CutRodAux}(p,\ n,\ r)$
  if $r[n] \geq 0$ **then**
    return $r[n]$
  end if
  if $n = 0$ **then**
    $q \leftarrow 0$
  else
    $q \leftarrow -\infty$
    for $i = 1$ to n **do**
      $q \leftarrow \max(q, p[i] + \textsc{CutRodAux}(p, n - i, r))$
    end for
    $r[n] \leftarrow q$
  end if
  return q
end function
Compute the solution: Call $\textsc{CutRod}(p, n)$

Rod Cutting Problem: Bottom-Up

Algorithm 2 Rod Cutting - Bottom Up

Require: p: Array of prices for each length, n: Length of the rod

Ensure: Maximum revenue obtainable for a rod of length n

function CUTRODBOTTOMUP(p, n)
    Initialize an array $r[0 \dots n]$ with $r[0] \leftarrow 0$
    for $j = 1$ to n **do**
        $q \leftarrow -\infty$
        for $i = 1$ to j **do**
            $q \leftarrow \max(q, p[i] + r[j - i])$
        end for
        $r[j] \leftarrow q$
    end for
    return $r[n]$
end function

Coin Change II: Maximum Number of Ways

- **Given:**
  - A set of coins with distinct denominations $\{c_1, c_2, \ldots, c_m\}$.
  - A target amount n.
- **Objective:** Find the maximum number of ways to make amount n using the given coins.
- Example:
  - Coins: $\{1, 2, 5\}$, Amount: $n = 5$
  - Output: 4 ways
    1. 5
    2. $2, 2, 1$
    3. $2, 1, 1, 1$
    4. $1, 1, 1, 1, 1$

Recall: Subset Sum Problem

- **Based on the 0/1 Knapsack Problem.**
- Given:
  - A set of integers, $\{a_1, a_2, \ldots, a_m\}$.
  - A target sum S.
- Objective: Determine if there exists a subset of integers that adds up to the target sum S.
- Example:
  - Set: $\{3, 34, 4, 12, 5, 2\}$
  - Target sum: $S = 9$
  - Output: Yes (subset $\{4, 5\}$ adds up to 9)

Recall: Subset Sum Problem

Approach:

- Define $dp[i][j]$ as `True` if a subset of the first i elements can sum to j, otherwise `False`.
- Recurrence Relation:

$$dp[i][j] = dp[i-1][j] \text{ or } dp[i-1][j-a_i]$$

  where a_i is the current element.
- Base Case:
  - $dp[0][0] = \text{True}$: A sum of 0 can be achieved with no elements.
  - For all $j > 0$: $dp[0][j] = \text{False}$.

Coin Change II: Parallel with Subset Sum Problem

- Both problems seek ways to reach a target sum or amount using given values.
- **Subset Sum:** Checks if there exists a subset with the target sum.
- **Coin Change:** Counts the number of ways to reach the target amount.
- Differences:
  - **Element Repetition:** In Subset Sum, each element is used only once. In Coin Change II, coins can be reused multiple times.
  - **Goal:** Subset Sum is about existence, while Coin Change II is about counting ways.

Adapting Subset Sum to Coin Change II

Subset Sum Recurrence Relation

$$dp[i][j] = dp[i-1][j] \text{ or } dp[i-1][j-a_i]$$

- **Interpretation:** Checks if it is possible to form sum j using the first i elements.
- **Explanation:** Each element can be used at most once, so we consider two options:
  - Exclude the current element a_i: $dp[i][j] = dp[i-1][j]$.
  - Include the current element a_i: $dp[i][j] = dp[i-1][j-a_i]$.

Adapting Subset Sum to Coin Change II

Coin Change II Recurrence Relation

$$dp[j] = dp[j] + dp[j - c_i]$$

- **Interpretation:** Counts the number of ways to make amount j by including coin c_i multiple times.
- **Explanation:** To account for reuse of coins:
  - Transition from a 2D `True/False` DP table to a 1D integer DP array for counts.
  - Add $dp[j - c_i]$ to $dp[j]$ to accumulate the number of ways.

- **Key Change:** Replace `True/False` checks with integer addition to count combinations, and allow repeated use of elements.

Coin Change II: Recurrence Relation

- Define $dp[j]$ as the number of ways to make amount j using the available coins.
- **Base Case:** $dp[0] = 1$ (1 way to make amount 0, by using no coins).
- For each coin c_i and each amount $j \geq c_i$:
  - Add the number of ways to make amount $j - c_i$ to $dp[j]$.
  - Formula:

$$dp[j] = dp[j] + dp[j - c_i]$$

- Intuition: $dp[j - c_i]$ represents ways to make j if we include coin c_i.

Coin Change II: Bottom-Up Algorithm

Algorithm 1 Coin Change: Maximum Number of Ways

Require: C: Array of coin denominations, n: Target amount

Ensure: Number of ways to make amount n

```
function CoinChangeII(C, n)
    Initialize dp[0...n] with dp[0] ← 1 and dp[j] ← 0 for j > 0
    for each coin c in C do
        for j = c to n do
            dp[j] ← dp[j] + dp[j − c]
        end for
    end for
    return dp[n]
end function
```

Coin Change II: Example

- **Coins:** $\{1, 2, 5\}$, **Target Amount:** $n = 5$
- Initial DP Array: $dp = [1, 0, 0, 0, 0, 0]$

Steps for Each Coin

Using Coin 1:

- For amount 1: $dp[1] = dp[1] + dp[1-1] = 0 + 1 = 1$
- For amount 2: $dp[2] = dp[2] + dp[2-1] = 0 + 1 = 1$
- For amount 3: $dp[3] = dp[3] + dp[3-1] = 0 + 1 = 1$
- For amount 4: $dp[4] = dp[4] + dp[4-1] = 0 + 1 = 1$
- For amount 5: $dp[5] = dp[5] + dp[5-1] = 0 + 1 = 1$

DP Array after using coin 1: $dp = [1, 1, 1, 1, 1, 1]$

Coin Change II: Example

- **Coins:** $\{1, 2, 5\}$, **Target Amount:** $n = 5$
- **DP Array after using coin 1:** $dp = [1, 1, 1, 1, 1, 1]$

Steps for Each Coin

Using Coin 2**:**

- For amount 2: $dp[2] = dp[2] + dp[2 - 2] = 1 + 1 = 2$
- For amount 3: $dp[3] = dp[3] + dp[3 - 2] = 1 + 1 = 2$
- For amount 4: $dp[4] = dp[4] + dp[4 - 2] = 1 + 2 = 3$
- For amount 5: $dp[5] = dp[5] + dp[5 - 2] = 1 + 2 = 3$

DP Array after using coin 2: $dp = [1, 1, 2, 2, 3, 3]$

Coin Change II: Example

- **Coins:** $\{1, 2, 5\}$, **Target Amount:** $n = 5$
- **DP Array after using coin 2:** $dp = [1, 1, 2, 2, 3, 3]$

Steps for Each Coin

Using Coin 5:

- For amount 5: $dp[5] = dp[5] + dp[5 - 5] = 3 + 1 = 4$

DP Array after using coin 5: $dp = [1, 1, 2, 2, 3, 4]$

Result: There are $dp[5] = 4$ ways to make amount 5 using coins $\{1, 2, 5\}$.