Introduction to Competitive Programming

Agenda

- Dynamic Programming with 2D tables
- Knapsack
- Coin Change without Repetition
- Longest common subsequence

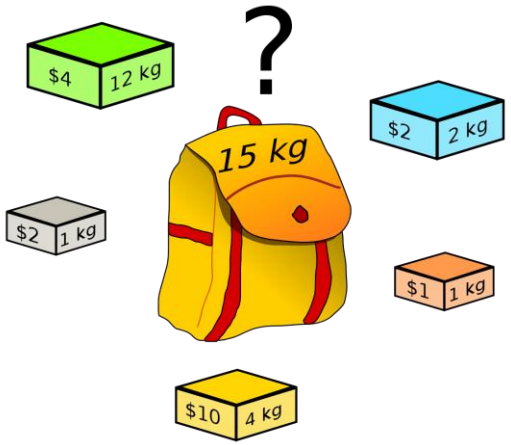
Example: Knapsack Problem

Given n items of

integer weights: w_1 w_2 ... w_n

values: v_1 v_2 ... v_n

a knapsack of integer capacity W



Find most valuable subset of the items that fit into the knapsack

Brute-force approach:

Consider all subsets of items (2^n of them!)

Compute the weight of all subsets

Compute the value for each *valid* subset

Return subset of maximum value

Knapsack Problem: Brute Force

Find the most valuable subset of the items that fit into a knapsack capacity W.

Example: Knapsack capacity W=16

item	weight	value
1	2	$20
2	5	$30
3	10	$50
4	5	$10

Knapsack by exhaustive search

Subset	Total weight	Total value
{1}	2	$20
{2}	5	$30
{3}	10	$50
{4}	5	$10
{1,2}	7	$50
{1,3}	12	$70
{1,4}	7	$30
{2,3}	**15**	**$80**
{2,4}	10	$40
{3,4}	15	$60
{1,2,3}	**17**	not feasible
{1,2,4}	12	$60
{1,3,4}	**17**	not feasible
{2,3,4}	**20**	not feasible
{1,2,3,4}	**22**	not feasible

item	weight	value
1	2	$20
2	5	$30
3	10	$50
4	5	$10

Knapsack – DP Example

n = 4 (items), weights = {2,1,3,2}, values = {12,10,20,15}

Knapsack capacity W = 5

Capacity j

i	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					GOAL

Knapsack – DP Subproblem ($i = 1$)

i = **1**, weight = {**2**}, value = {**12**}

Knapsack capacity j = 1, 2, …, 5

$$V[1, j] = \text{maximum of: } \begin{cases} V[\text{i-1}, j] \\ 12 + V[0, j\text{-}2] \quad \text{If j>=w} \end{cases}$$

Capacity j

i	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$
0	0	0	0	0	0	0
1	0					

w_1=2, v_1=12

Knapsack – DP Subproblem ($i = 2$)

i = **2**, weights = {2, **1**}, values = {12, **10**}

Knapsack capacity j = 1, 2, …, 5

$$V[\mathbf{2}, j] = \text{maximum of: } \begin{cases} V[\text{i-1}, j] \\ \mathbf{10} + V[1, j\text{-}\mathbf{1}] \end{cases} \quad \text{If w>=j}$$

Capacity j

i	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0					

w_2=1, v_2=10

Knapsack – DP Subproblem ($i = 3$)

i = **3**, weights = {2, 1, **3**}, values = {12, 10, **20**}

Knapsack capacity j = 1, 2, …, 5

$$V[3, j] = \text{maximum of: } \begin{cases} V[\text{i-1}, j] \\ \mathbf{20} + V[2, j\text{-}3] \quad \text{If w>=j} \end{cases}$$

Capacity j

i	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0					

w_3=3, v_3=20

Knapsack – DP Subproblem ($i = 4$)

i = **4**, weights = [2, 1, 3, **2**], values = [12, 10, 20, **15**]

Knapsack capacity j = 1, 2, …, 5

$$V[\mathbf{4}, j] = \text{maximum of: } \begin{cases} V[\text{i-1}, j] \\ \mathbf{15} + V[2, j - \mathbf{2}] \quad \text{If w>=j} \end{cases}$$

Capacity j

i	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

w_4=2, v_4=15

Backtrack to get items that make up most valuable subset

Use the recurrence:

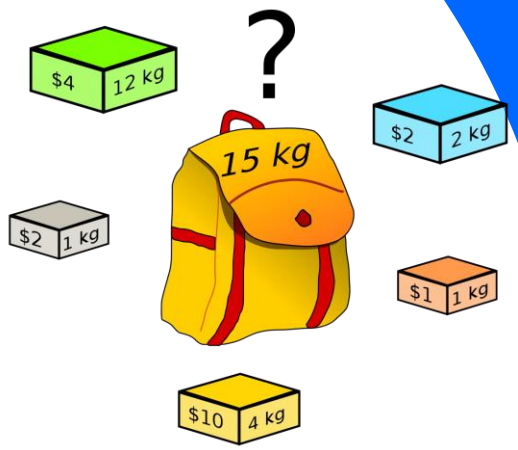
$V[n,W]$ = maximum of:

$$\begin{cases} V[n\text{-}1, W] \\ v_n + V[n\text{-}1, W\text{-}w_n] \end{cases}$$

Check if V[n,w]=V[n-1,w] if it does then we know the last item is not included.
If it doesn't then the last item was included.
Continue to backtrack to find the next item, by looking at
$V[n\text{-}1,w\text{-}w_n]$

Repeat this process to find all items included.

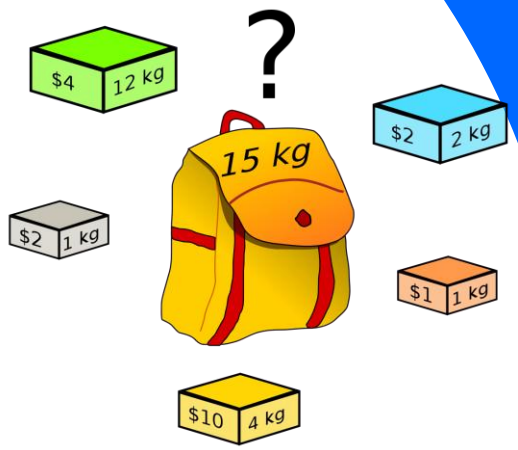
Fun Fact – Knapsack with Repetition



The knapsack algorithm on the previous slides solves the problem assuming an item can only be used once. ***Knapsack without repetition.***

There is a version of the problem that allows for repetition. To solve knapsack with repetition, use a 1D array like in coin change, and tweak the base case and recurrence. ***Try it!***

Fun Fact – Coin Change without Repetition



The classic coin change problem assumes an infinite number of each denomination of coins.

There is a version of the coin change problem that does not allow the repetition of coins. Tweak knapsack without repetition to solve it. Must change the base case to inf since we are finding the minimum here and we want exact change. ***Try it!***

You can use this algorithm to solve the kattis problem ninepacks.

Coin Change Without Repetition

Coins=[2,3,5] target = 7

Notice the base case for the first row is inf. Knapsack is 0.
We are finding the minimum value not the max. We also want exact change.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	**0**	inf	inf	inf	inf	inf	inf	inf
1 (coin 2)	**0**	inf	1	inf	inf	inf	inf	inf
2 (coin 3)	**0**	inf	1	1	inf	2	inf	inf
3 (coin 5)	**0**	inf	1	1	inf	1	inf	2

```
def min_coins_to_make_target(n, coins, target):
    # DP table: (n+1) x (target+1), initialized to a large value
    INF = float('inf')
    dp = [[INF] * (target + 1) for _ in range(n + 1)]

    # Base Case: It takes 0 coins to make sum 0
    for i in range(n + 1):
        dp[i][0] = 0

    # Fill the DP table
    for i in range(1, n + 1):
        for j in range(target + 1):
            # Option 1: Exclude the current coin (inherit from previous row)
            dp[i][j] = dp[i-1][j]

            # Option 2: Include the coin (if possible)
            if j >= coins[i-1]:
                dp[i][j] = min(dp[i][j], dp[i-1][j - coins[i-1]] + 1)
```

Longest Common Subsequence (LCS)

Definition: The Longest Common Subsequence (LCS) of two sequences is the longest subsequence that appears in both sequences in the same order but not necessarily consecutively.

- Example:
  - Sequence 1: ACDBE
  - Sequence 2: ABCDE
  - LCS: ACDE (appears in both in order)

ABE is also a common subsequence but not the longest

ABC is not a common subsequence

Longest Common Subsequence (LCS)

Applications:

- DNA sequence alignment
- File comparison (diff tools)
- Text similarity detection

LCS Example Calculation

Given X = "AGGTAB", Y = "GXTXAYB"

DP Table Calculation:

	0	**G**	**X**	**T**	**X**	**A**	**Y**	**B**
0		0	0	0	0	0	0	0
A	0	0	0	0	0	1	1	1
G	0	1	1	1	1	1	1	1
G	0	1	1	1	1	1	1	1
T	0	1	1	2	2	2	2	2
A	0	1	1	2	2	3	3	3
B	0	1	1	2	2	3	3	4

LCS = "GTAB" (length = 4)

Note the LCS does not have to end at the last character.

The LCS can occur anywhere within the strings.

LCS Algorithm:

1. Dynamic programming table:
    - Initialize a 2D table, T to hold the current LCS index place i,j
    - T[i][j] = holds the LCS considering substrings X[0..i] and Y[0..j].
    - T[0][j]=0 for all 0<=j<=n where n=len(X)
    - T[i][0]=0 for all 0<=j<=m where m=len(Y)

2.Iterate through the table 1..n:

    Iterate through table 1..m:
    - Check if the characters at X[i-1] = Y[j-1]
    - If yes, add 1 to the number of matching characters found at t[i-1][j-1] (this is the number of matches excluding the current character)
    - Otherwise take the max of t[i][j-1] and t[i-1][j]
    - The result is stored at t[n][m].

Time Complexity:

- $O(n^2)$ (single pass through the 2D array)

LCS Using Dynamic Programming

• Define dp[i][j] as the LCS length of first i characters of string X and first j characters of string Y.

• Recurrence Relation:

- If X[i-1] == Y[j-1]:

  dp[i][j] = dp[i-1][j-1] + 1

- Else:

  dp[i][j] = max(dp[i-1][j], dp[i][j-1])

• Base Case: dp[0][j] = 0 and dp[i][0] = 0