Dynamic Programming

CS 4104: Data and Algorithm Analysis

Yoseph Berhanu Alebachew

May 11, 2025

Virginia Tech

Table of contents

Introduction

Techniques Discussed so far

- We began our study of algorithmic techniques with greedy algorithms.
- Greedy algorithms form the most natural approach to algorithm design.
- Challenge: Determine whether a proposed greedy algorithm provides a correct solution in all cases.
- The problems we discussed earlier in the course had a greedy algorithm that worked.
- This is not true in general; most problems lack a natural greedy algorithm.

Techniques Discussed so far

- For such problems, other approaches are necessary.
- Divide and conquer can sometimes serve as an alternative approach.
- However, it often isn't strong enough to reduce exponential brute-force search to polynomial time.
- We now turn to a more powerful and subtle design technique: dynamic programming.
- Basic idea drawn from the intuition behind divide and conquer.
- Opposite of the greedy strategy: explores the space of all possible solutions by decomposing into sub-problems.

Dynamic Programming: Introduction

- Dynamic Programming (DP) is a method for solving complex problems by breaking them down into simpler sub-problems.
- In Divide and Conquer where sub-problems are independent of each other
- DP is applicable when the problem can be divided into **overlapping sub-problems**
- In DP we avoid having to recompute solutions for the same sub-problem more than once.
- Another key property in DP is the optimal substructure, as we discussed in Greedy Algorithm.
- Optimal substructure is a property of a problem that indicates that an optimal solution to the problem can be constructed efficiently from optimal solutions to its sub-problems

Dynamic Programming: Introduction

- Key Characteristics
  - Carefully decomposes things into a series of sub-problems.
  - Builds up correct solutions to larger and larger sub-problems.
  - Operates close to the edge of brute-force search, but avoids examining all solutions explicitly.
- Challenges
  - It is a tricky technique to get used to.
  - Requires practice to become comfortable with it.
- Two approaches
  - As a recursive procedure resembling brute-force search, but we memoize solutions
  - As an iterative algorithm building up solutions to larger sub-problems.

Weighted Interval Scheduling

Weighted Interval Scheduling: Introduction

- Given a set of intervals, each with a start time, end time, and weight.
- We want to select a subset of non-overlapping intervals with the maximum **total weight.**
- Remember in Interval Scheduling Problem:
  - Accept the largest set of non-overlapping intervals.
  - All intervals have the same value
  - Efficient greedy algorithm exists
- In Weighted Interval Scheduling
  - Each interval has a weight
  - Intervals have different values
  - Focus is on maximizing accepted weight not count

Weighted Interval Scheduling Problem: Problem Statement

- A more formal *problem statement* of weighted interval scheduling is

Weighted Interval Scheduling Problem: Problem Statement

- A more formal *problem statement* of weighted interval scheduling is
  - Input: n requests labeled 1, ..., n. Each request i has a start time s_i, finish time f_i, and value v_i.

Weighted Interval Scheduling Problem: Problem Statement

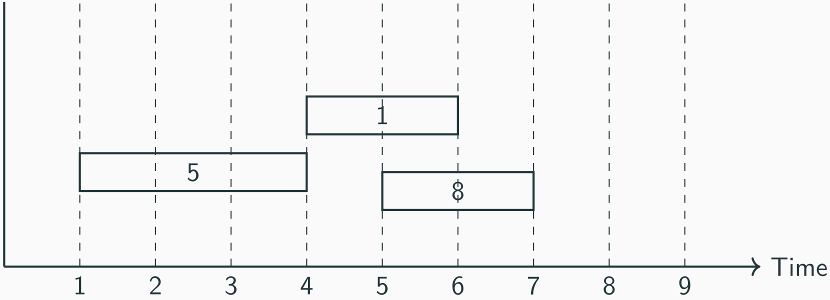
- A more formal *problem statement* of weighted interval scheduling is
  - Input: n requests labeled 1, ..., n. Each request i has a start time s_i, finish time f_i, and value v_i.
  - Goal: select a subset S of mutually compatible intervals to maximize $\sum_{i \in S} v_i$.

Weighted Interval Scheduling Problem: Problem Statement

- A more formal *problem statement* of weighted interval scheduling is
  - Input: n requests labeled 1, ..., n. Each request i has a start time s_i, finish time f_i, and value v_i.
  - Goal: select a subset S of mutually compatible intervals to maximize $\sum_{i \in S} v_i$.
- Consider the example

Weighted Interval Scheduling Problem: Problem Statement

- A more formal *problem statement* of weighted interval scheduling is
  - Input: n requests labeled 1, ..., n. Each request i has a start time s_i, finish time f_i, and value v_i.
  - Goal: select a subset S of mutually compatible intervals to maximize $\sum_{i \in S} v_i$.
- Consider the example
  - $A = [(1,4,5),(4,6,1),(5,7,8)]$



WIS: Greedy Algorithm

- Greedy algorithms do not consider weights of intervals.
- Example:
  - Interval A: (1, 4), weight $= 5$
  - Interval B: (4, 6), weight $= 1$
  - Interval C: (5, 7), weight $= 8$
- Greedy choice of earliest finish: selects B and C (weight $= 6$)
- The optimal choice was A and C (weight $= 13$).
- **Questions:** Poll 1
  - Can we modify the greedy choice strategy to solve this ? How?
  - Can we use divide and conquer to find efficient algorithm for this problem ?

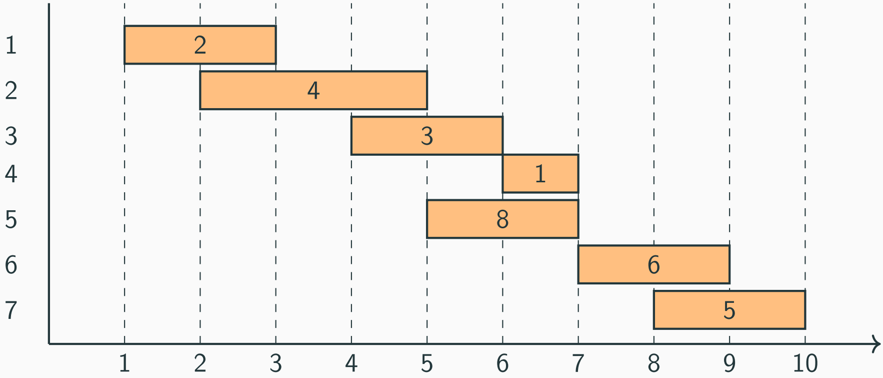
WIS: Brute Force Approach

- **Problem Statement**:
  - Given a set of intervals, each with a start time, end time, and weight, find a subset of non-overlapping intervals with the maximum total weight.
- **Brute Force Approach**:
  - **Step 1**: Enumerate all subsets of the given intervals.
  - **Step 2**: Check each subset for overlapping intervals.
  - **Step 3**: Calculate the total weight of each subset.
  - **Step 4**: Select the subset with the maximum total weight.
- **Complexity Analysis**:
  - The number of subsets is 2^n, where n is the number of intervals.
  - Checking for overlaps and calculating the weight takes $O(n)$ time per subset.
  - Total time complexity is $O(n \cdot 2^n)$.

WIS: Recursive Solution

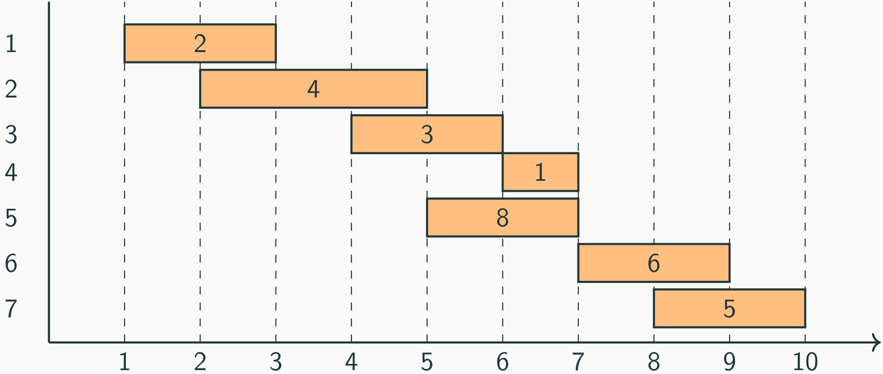
- Sort intervals by finish times.
- Now we define a utility procedure called $p(j)$
- $p(j)$ returns the largest index $i < j$ such that interval i and j are compatible.
- Consider the example:
  $A = [(4, 6, 3), (2, 5, 4), , (5, 7, 8), (1, 3, 2), (6, 8, 1)(8, 10, 5), (7, 9, 6)]$
- We first sort the intervals by finish time
  $A = [(1, 3, 2), (2, 5, 4), (4, 6, 3), (6, 7, 1), (5, 7, 8), (7, 9, 6), (8, 10, 5)]$

WIS: Recursive Solution



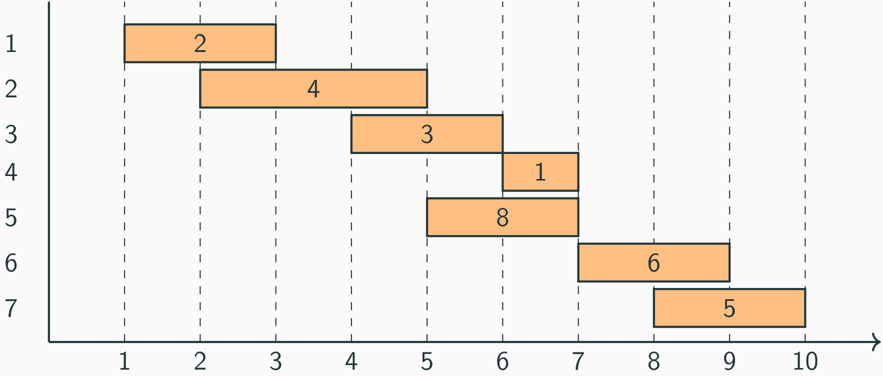
$p(1) =$

WIS: Recursive Solution



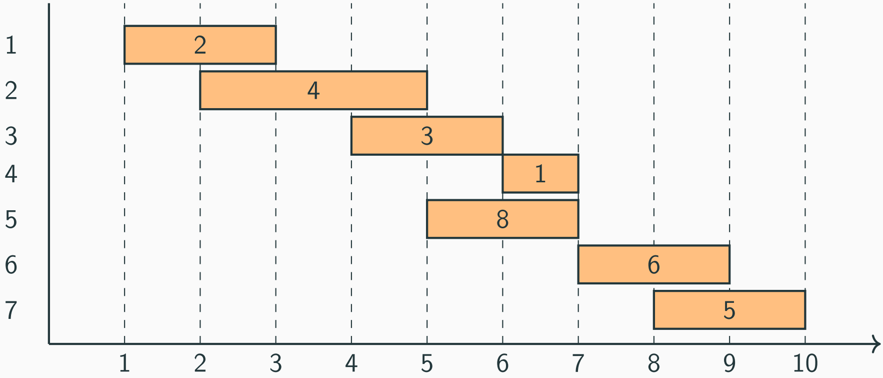
$p(1) = 0$

WIS: Recursive Solution



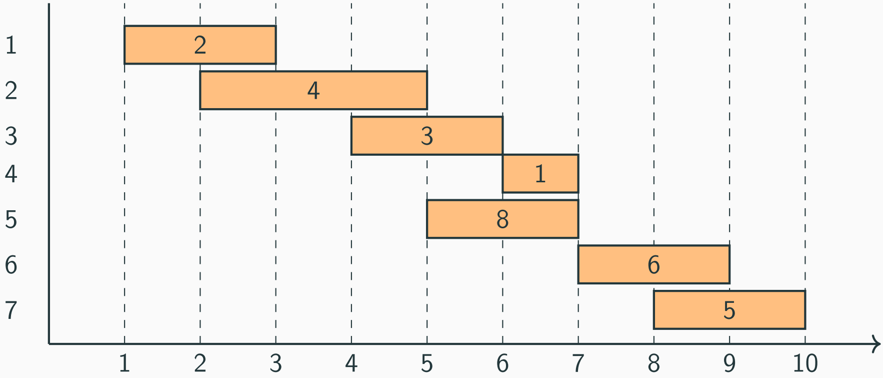
$p(2) =$

WIS: Recursive Solution



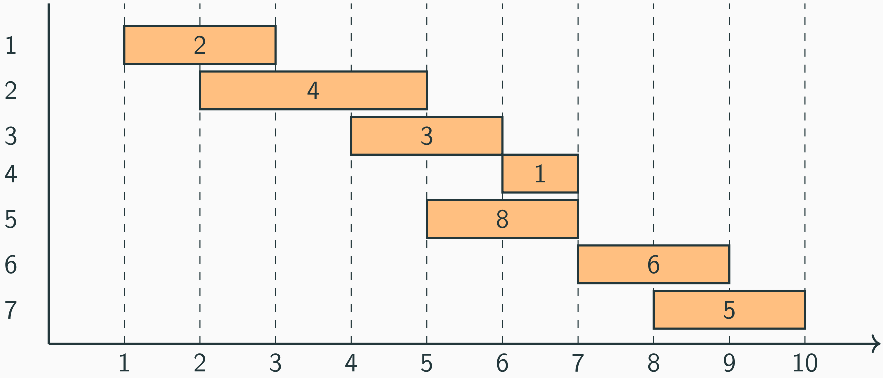
$p(2) = 0$

WIS: Recursive Solution



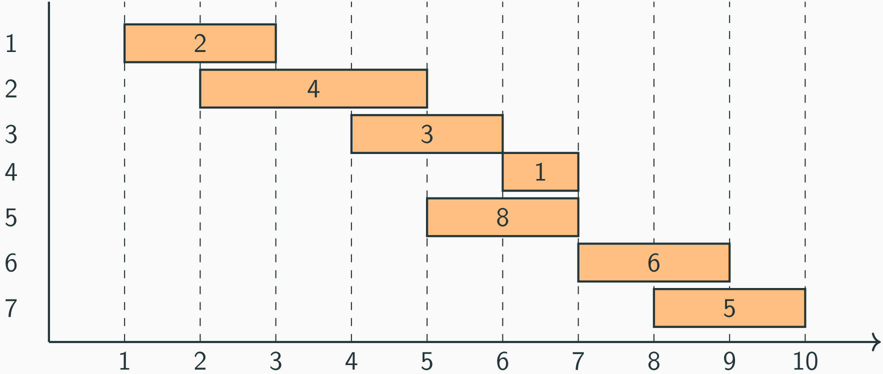
$$p(3) =$$

WIS: Recursive Solution



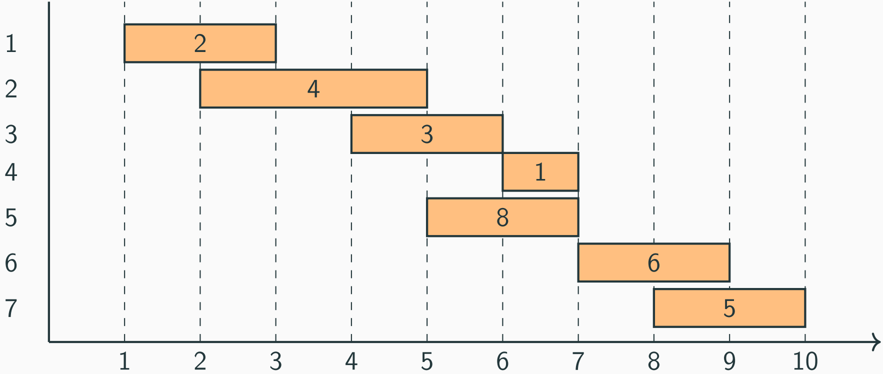
$$p(3) = 1$$

WIS: Recursive Solution



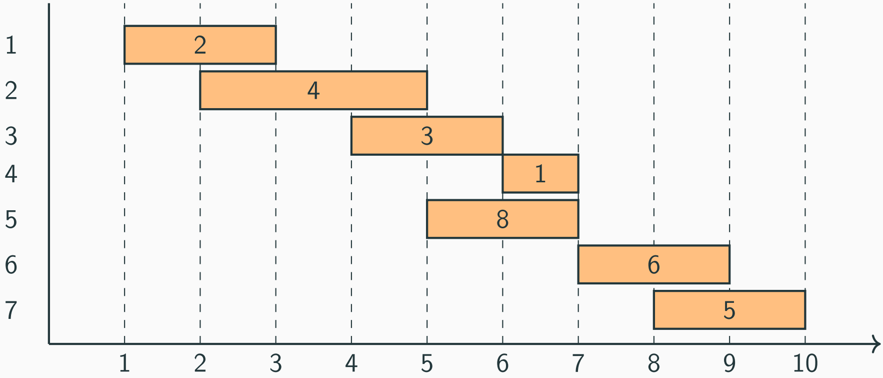
$$\boxed{p(4) =}$$

WIS: Recursive Solution



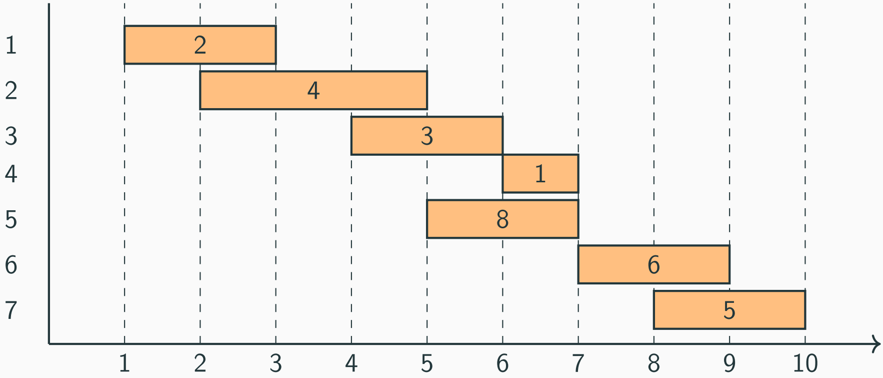
$p(4) = 3$

WIS: Recursive Solution



$p(5) =$ Poll 2

WIS: Recursive Solution



$p(5) = 2$

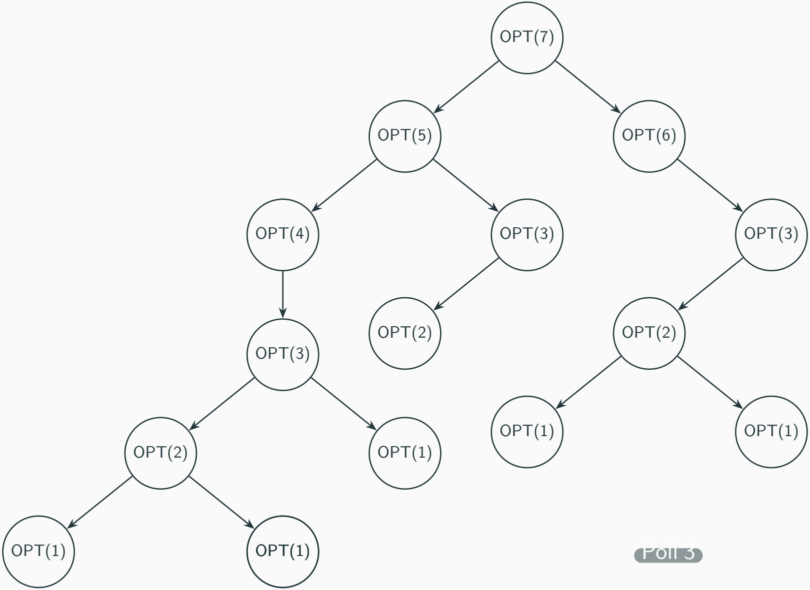
WIS: Recursive Solution

- Given the definition of $p(j)$
- Define $OPT(j)$ as the maximum weight of a subset of intervals $\{1, 2, ..., j\}$.
- Recursive formula:

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max(v_j + OPT(p(j)), OPT(j-1)) & \text{otherwise} \end{cases}$$

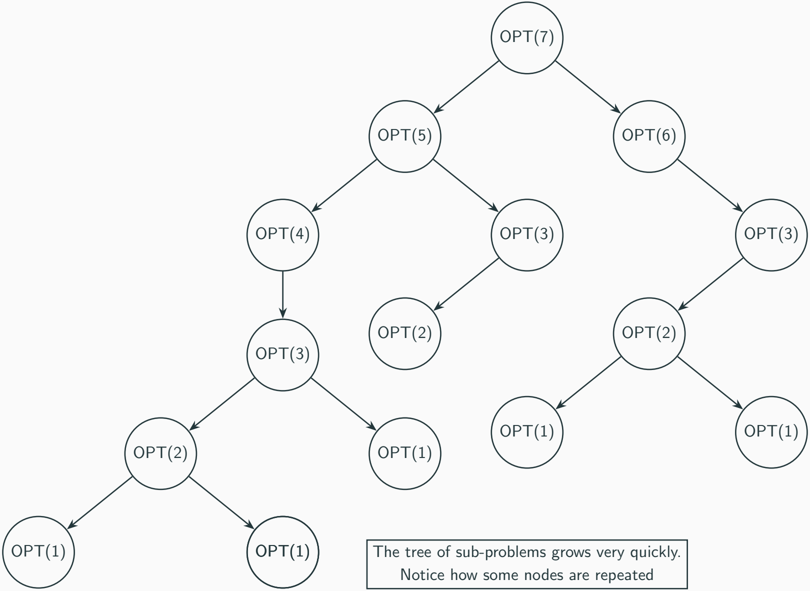
- Here v_j is the value we get for choosing to include interval j
- Notice the recursion and what we need to compute at each stage

WIS: Recursive Solution - Visualization



Poll 3

WIS: Recursive Solution - Visualization



The tree of sub-problems grows very quickly.
Notice how some nodes are repeated

WIS: Dynamic Programming

- The key idea behind Dynamic Programming(DP) is systematically avoid computing solutions for a sub problem more than once
- This can be achieved by storing and reusing solutions to sub-problems.
  - We can do this bottom-up or top-down
- In top down, a.k.a Memoization, we take a not of the sub-problems we solved
- If we encounter the same sub-problem again we reuse the solution without recomputing
- In the bottom up approach we iterate through the sub-problems and solve them incrementally
- We use tabulation to achieve this
- The table is filled iteratively from the smallest sub-problem to the largest, ensuring that each sub-problem is solved only once.

WIS: Dynamic Programming Example

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max(v_j + OPT(p(j)), OPT(j-1)) & \text{otherwise} \end{cases}$$

j	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_j	0	2	4	3	1	8	6	5
$p(j)$	0							
$OPT(j)$								

WIS: Dynamic Programming Example

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max(v_j + OPT(p(j)), OPT(j-1)) & \text{otherwise} \end{cases}$$

j	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_j	0	2	4	3	1	8	6	5
$p(j)$	0	0						
$OPT(j)$								

WIS: Dynamic Programming Example

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max(v_j + OPT(p(j)), OPT(j-1)) & \text{otherwise} \end{cases}$$

j	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_j	0	2	4	3	1	8	6	5
$p(j)$	0	0	0					
$OPT(j)$								

WIS: Dynamic Programming Example

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max(v_j + OPT(p(j)), OPT(j-1)) & \text{otherwise} \end{cases}$$

---

j	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_j	0	2	4	3	1	8	6	5
$p(j)$	0	0	0	1				
$OPT(j)$								

WIS: Dynamic Programming Example

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max(v_j + OPT(p(j)), OPT(j-1)) & \text{otherwise} \end{cases}$$

j	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_j	0	2	4	3	1	8	6	5
$p(j)$	0	0	0	1	3			
$OPT(j)$								

WIS: Dynamic Programming Example

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max(v_j + OPT(p(j)), OPT(j-1)) & \text{otherwise} \end{cases}$$

j	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_j	0	2	4	3	1	8	6	5
$p(j)$	0	0	0	1	3	2		
$OPT(j)$								

WIS: Dynamic Programming Example

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max(v_j + OPT(p(j)), OPT(j-1)) & \text{otherwise} \end{cases}$$

---

j	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_j	0	2	4	3	1	8	6	5
$p(j)$	0	0	0	1	3	2	5	
$OPT(j)$								

WIS: Dynamic Programming Example

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max(v_j + OPT(p(j)), OPT(j-1)) & \text{otherwise} \end{cases}$$

j	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_j	0	2	4	3	1	8	6	5
$p(j)$	0	0	0	1	3	2	5	5
$OPT(j)$								

WIS: Dynamic Programming Example

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max(v_j + OPT(p(j)), OPT(j-1)) & \text{otherwise} \end{cases}$$

$$OPT(0) = 0$$

j	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_j	0	2	4	3	1	8	6	5
$p(j)$	0	0	0	1	3	2	5	5
$OPT(j)$	0							

WIS: Dynamic Programming Example

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max(v_j + OPT(p(j)), OPT(j-1)) & \text{otherwise} \end{cases}$$

$$OPT(1) = \max(v_1 + OPT(p(1)), OPT(0)) = \max(2 + 0, 0) = 2$$

j	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_j	0	2	4	3	1	8	6	5
$p(j)$	0	0	0	1	3	2	5	5
$OPT(j)$	0	2						

WIS: Dynamic Programming Example

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max(v_j + OPT(p(j)), OPT(j-1)) & \text{otherwise} \end{cases}$$

$$OPT(2) = \max(v_2 + OPT(p(2)), OPT(1)) = \max(4 + 0, 2) = 4$$

j	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_j	0	2	4	3	1	8	6	5
$p(j)$	0	0	0	1	3	2	5	5
$OPT(j)$	0	2	4					

WIS: Dynamic Programming Example

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max(v_j + OPT(p(j)), OPT(j-1)) & \text{otherwise} \end{cases}$$

$$OPT(3) = \max(v_3 + OPT(p(3)), OPT(2)) = \max(3 + 2, 4) = 5$$

j	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_j	0	2	4	3	1	8	6	5
$p(j)$	0	0	0	1	3	2	5	5
$OPT(j)$	0	2	4	5				

WIS: Dynamic Programming Example

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max(v_j + OPT(p(j)), OPT(j-1)) & \text{otherwise} \end{cases}$$

$$OPT(4) = \max(v_4 + OPT(p(4)), OPT(3)) = \max(1 + 5, 5) = 6$$

j	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_j	0	2	4	3	1	8	6	5
$p(j)$	0	0	0	1	3	2	5	5
$OPT(j)$	0	2	4	5	6			

WIS: Dynamic Programming Example

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max(v_j + OPT(p(j)), OPT(j-1)) & \text{otherwise} \end{cases}$$

$$OPT(5) = \max(v_5 + OPT(p(5)), OPT(4)) = \max(8 + 4, 5) = 12$$

j	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_j	0	2	4	3	1	8	6	5
$p(j)$	0	0	0	1	3	2	5	5
$OPT(j)$	0	2	4	5	6	12		

WIS: Dynamic Programming Example

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max(v_j + OPT(p(j)), OPT(j-1)) & \text{otherwise} \end{cases}$$

$$OPT(6) = ?, OPT(7) = ?$$

Poll 4

j	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_j	0	2	4	3	1	8	6	5
$p(j)$	0	0	0	1	3	2	5	5
$OPT(j)$	0	2	4	5	6	12		

WIS: Dynamic Programming Example

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max(v_j + OPT(p(j)), OPT(j-1)) & \text{otherwise} \end{cases}$$

$$OPT(6) = \max(v_6 + OPT(p(6)), OPT(5)) = \max(6 + 12, 12) = 18$$

j	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_j	0	2	4	3	1	8	6	5
$p(j)$	0	0	0	1	3	2	5	5
$OPT(j)$	0	2	4	5	6	12	18	

WIS: Dynamic Programming Example

$$OPT(7) = \max(v_7 + OPT(p(7)), OPT(6)) = \max(5 + 12, 18) = 18$$

j	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_j	0	2	4	3	1	8	6	5
$p(j)$	0	0	0	1	3	2	5	5
$OPT(j)$	0	2	4	5	6	12	18	18

WIS: Dynamic Programming Example

j	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_j	0	2	4	3	1	8	6	5
$p(j)$	0	0	0	1	3	2	5	5
$OPT(j)$	0	2	4	5	6	12	18	18

- The maximum value we can get is 18
- We can trace back the table to figure out with intervals lead to this value
- The intervals are $\{6, 5, 2\}$

Principles of Dynamic Programming

Principles of Dynamic Programming: Algorithm Design

- Memoization or Iteration over Sub-problems
- We'll use the Weighted Interval Scheduling Problem to summarize the basic principles of dynamic programming.
- Focus on iterating over sub-problems, rather than computing solutions recursively.
- The key to the efficient algorithm is the array/table we fill, lets call it M.
- M encodes the value of optimal solutions to sub-problems on intervals $\{1, 2, \ldots, j\}$ for each j.
- Using M, the problem is solved: $M[n]$ contains the value of the optimal solution for the full instance.
- The new formulation explicitly captures the essence of dynamic programming and serves as a general template.

Principles of Dynamic Programming: Designing the Algorithm

- The key component is the array M:
  - $M[j]$ is defined based on values earlier in the array using:

$$M[j] = \max(v_j + M[p(j)], M[j-1])$$

- We can directly compute the entries in M by an iterative algorithm.
- The iterative algorithm:

```
Iterative-Compute-Opt
M[0] = 0
For j = 1, 2, ..., n
    M[j] = max(vj + M[p(j)], M[j-1])
Endfor
```

Analyzing the Algorithm

- By induction on j, we can prove that this algorithm writes $OPT(j)$ in array entry $M[j]$.
- The running time of Iterative-Compute-Opt is $O(n)$ since it runs for n iterations, spending constant time in each.
- Once the array M is filled, we can define a procedure, $Find - Solution$, that can trace back through M efficiently to return an optimal solution.
- Time complexity:
  - $O(n \log n)$ for sorting
  - $O(n \log n)$ for calculating $p(j)$
  - $O(n)$ for computing OPT / filling M.
  - $O(n)$ for $Find - Solution$.
  - Total $O(n \log n) + O(n \log n) + O(n) + O(n) = O(n \log n)$

Segmented Least Squares

Segmented Least Squares: The Problem

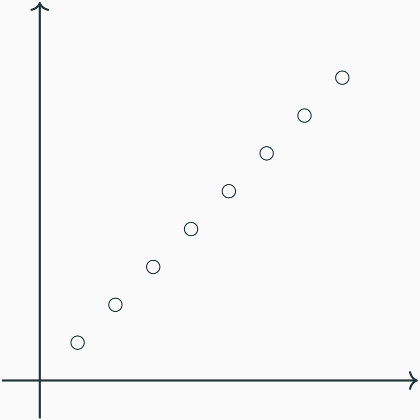
- Often when looking at scientific or statistical data, one tries to pass a "line of best fit" through the data.
- Given a set of points P in the plane, denoted $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \ldots < x_n$.
- We want to find a line L defined by $y = ax + b$ that minimizes the error:

$$\text{Error}(L, P) = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$

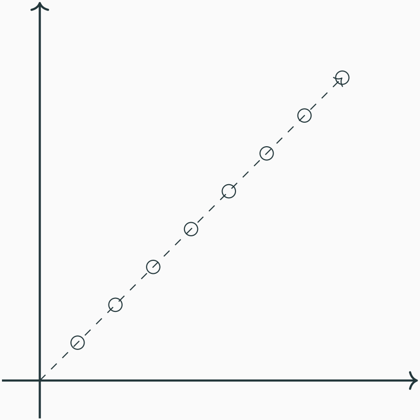
- The line of minimum error has a closed-form solution using calculus:

$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}, \quad b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

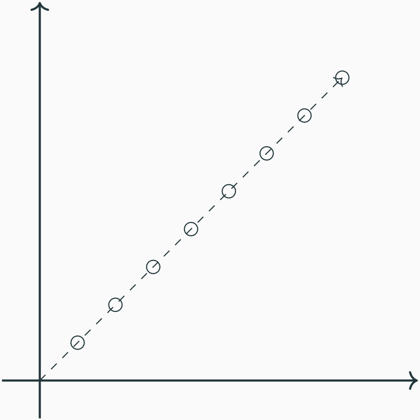
Segmented Least Squares: Example

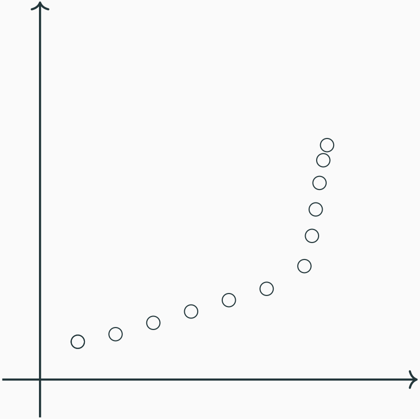


Segmented Least Squares: Example

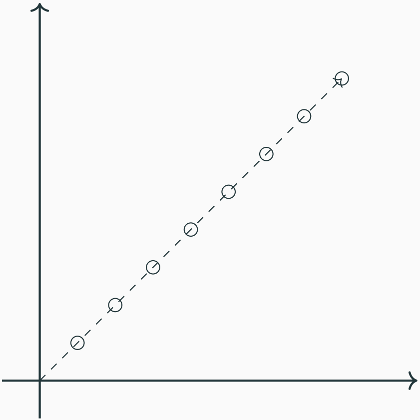


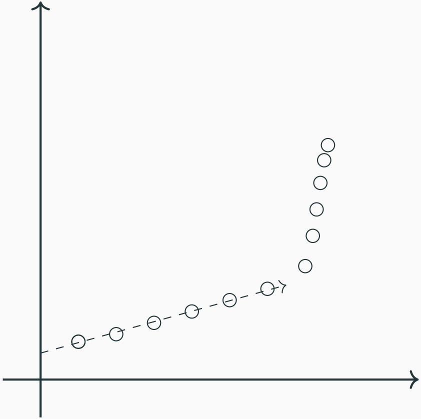
Segmented Least Squares: Example



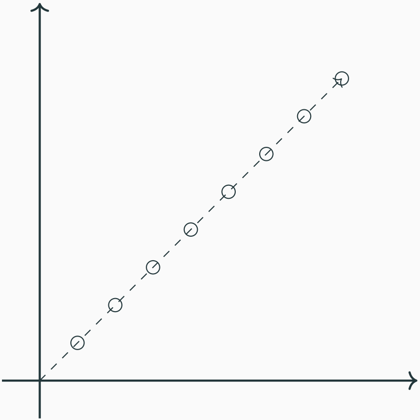


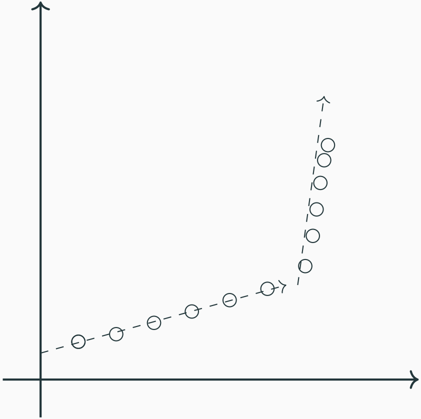
Segmented Least Squares: Example





Segmented Least Squares: Example





Segmented Least Squares: Introduction

- Sometimes data looks like it lies roughly on a sequence of lines.
- When the points are more complex, a single line may have a terrible error
- Multiple lines can achieve a smaller error.
- We aim to fit such points with as few lines as possible while minimizing the error.
- The more lines we use the lower the error
- We, of course don't want to use too many lines
- Let's define some penalty which we incur for every new line we introduce

Segmented Least Squares: Problem Statement

- We formalize this with the Segmented Least Squares Problem.
  - Given a set of points P, partition it into segments.
  - Each segment has a line minimizing the error.
  - Minimize the **penalty**,
- More formally:
  - Given points $P = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ with $x_1 < x_2 < \ldots < x_n$.
  - Partition P into segments $S = \{p_i, p_{i+1}, \ldots, p_j\}$ with minimum penalty.
  - Penalty includes the number of segments times a fixed multiplier $C > 0$ and the error of each segment.
    - C will be an input to our algorithm and controls how frugal we are when it comes to the number of segments to have
    - The **error** of a segment is the error value of the optimal line through each segment.

Segmented Least Squares: Recursive

- Let say, in our optimal solution, the last point, p_n is part of a segment that starts at p_i
- Let's also define our error for this segment as $e_{n,i}$
- Our penalty for the segment will become $e_{n,i} + C$
- We can now define the optimal solution for $OPT(n)$ as

$$OPT(n) = \min_{1 \leq i \leq n}(e_{i,n} + C + \mathrm{OPT}(i-1))C$$

- This means we recursively different starting point for our segment containing point n
- For any point j the we'll have the recurrence:

$$\mathrm{OPT}(j) = \min_{1 \leq i \leq j}(e_{i,j} + C + \mathrm{OPT}(i-1))$$

Segmented Least Squares: Dynamic Programming Properties

- Optimal substructure: The optimal solution to a problem contains the optimal solutions to its sub-problems.
- Overlapping sub-problems: The sub-problems are solved multiple times, which can be optimized using dynamic programming.
- The sub-problem $OPT(j)$ is built from the solutions of the smaller sub-problems $OPT(i-1)$, combined with the segment error $e_{i,j}$ and the penalty C.
- This allows us to use a bottom-up approach to fill in a table that stores the solutions to sub-problems.

Segmented Least Squares: Dynamic Programming Algorithm

Algorithm 1 Segmented-Least-Squares(n)

Array M[0 . . . n]
Set M[0] = 0
for all pairs i j **do**
    Compute the least squares error $e_{i,j}$ for the segment $p_i, \ldots, p_j$
end for
for j = 1, 2, . . . , n **do**
    Use the recurrence to compute M[j]
end for
return M[n]

Segmented Least Squares: Dynamic Programming

Algorithm 2 Find-Segments(j)

```
if j = 0 then
    Output nothing
else
    Find an i that minimizes e_{i,j} + C + M[i − 1]
    Output the segment {p_i, ..., p_j} and the result of Find-Segments(i-1)
end if
```

Segmented Least Squares: Analyzing the Algorithm

- Compute the least squares errors $e_{i,j}$ in $O(n^3)$ time:
  - There are $O(n^2)$ pairs (i, j).
  - Computing $e_{i,j}$ for each pair takes $O(n)$ time.
- The dynamic programming algorithm has n iterations, each taking $O(n)$ time:
  - For each j, finding the minimum using the recurrence takes $O(n)$ time.
  - Therefore, filling the array M takes $O(n^2)$ time.
- Thus, the overall running time is $O(n^3)$.

0/1 Knapsack Problem

0/1 Knapsack Problem: Definition

- Given items with weights and values, and a knapsack with a weight limit.
- Maximize the total value of items in the knapsack without exceeding the weight limit.
- Unlike fractional knapsack, here you either take an item in full or you leave it.
- Formal Problem statement:
  - **Input:**
    - A set of n items, each with a weight w_i and value v_i
    - A maximum weight capacity W of the knapsack
  - **Output:**
    - The maximum value achievable with a weight limit W

Note

Your text book presents a special case of knapsack as reformulated as a scheduling problem.

0/1 Knapsack Problem: Example

- Given the following items:
  - **Item 1:** Value $= 2$, Weight $= 3$
  - **Item 2:** Value $= 2$, Weight $= 1$
  - **Item 3:** Value $= 4$, Weight $= 3$
  - **Item 4:** Value $= 5$, Weight $= 4$
  - **Item 5:** Value $= 3$, Weight $= 2$
- Knapsack capacity $W = 7$

0/1 Knapsack Problem: Example

- Given the following items:
  - **Item 1:** Value $= 2$, Weight $= 3$
  - **Item 2:** Value $= 2$, Weight $= 1$
  - **Item 3:** Value $= 4$, Weight $= 3$
  - **Item 4:** Value $= 5$, Weight $= 4$
  - **Item 5:** Value $= 3$, Weight $= 2$
- Knapsack capacity $W = 7$
- Goal: Determine the maximum value that can be put into the knapsack.

0/1 Knapsack Problem: Example

- Given the following items:
  - **Item 1:** Value $= 2$, Weight $= 3$
  - **Item 2:** Value $= 2$, Weight $= 1$
  - **Item 3:** Value $= 4$, Weight $= 3$
  - **Item 4:** Value $= 5$, Weight $= 4$
  - **Item 5:** Value $= 3$, Weight $= 2$
- Knapsack capacity $W = 7$
- Goal: Determine the maximum value that can be put into the knapsack.
- Solution: The maximum value is 10, achieved by taking items 2, 4, and 5.

0/1 Knapsack Problem: Greedy Solution

- Greedy Approach:
  - Sort items by value-to-weight ratio (value/weight).
  - Item 2: Value = 2, Weight = 1, Ratio = 2.0
  - Item 5: Value = 3, Weight = 2, Ratio = 1.5
  - Item 3: Value = 4, Weight = 3, Ratio = 1.33
  - Item 1: Value = 2, Weight = 3, Ratio = 0.67
  - Item 4: Value = 5, Weight = 4, Ratio = 1.25
- Greedy Solution:
  - Take items 2, 5, and 3 (total weight = 6, total value = 9).
  - But knapsack capacity is not fully utilized.
  - Greedy approach does not always provide the optimal solution.

0/1 Knapsack Problem: Brute-force

- Brute-force Approach:
    - Explore all possible combinations of items.
    - Evaluate the total weight and value of each combination.
    - Select the combination with the maximum value that fits within the knapsack capacity.

0/1 Knapsack Problem: Brute-force

- Brute-force Approach:
    - Explore all possible combinations of items.
    - Evaluate the total weight and value of each combination.
    - Select the combination with the maximum value that fits within the knapsack capacity.
- Time Complexity:

0/1 Knapsack Problem: Brute-force

- Brute-force Approach:
  - Explore all possible combinations of items.
  - Evaluate the total weight and value of each combination.
  - Select the combination with the maximum value that fits within the knapsack capacity.
- Time Complexity:
  - Brute-force approach has a time complexity of $O(2^n)$, where n is the number of items.

0/1 Knapsack Problem: Brute-force

- Brute-force Approach:
  - Explore all possible combinations of items.
  - Evaluate the total weight and value of each combination.
  - Select the combination with the maximum value that fits within the knapsack capacity.
- Time Complexity:
  - Brute-force approach has a time complexity of $O(2^n)$, where n is the number of items.
  - Impractical for large n due to exponential growth.

0/1 Knapsack Problem: Dynamic Programming Algorithm

- Define $dp[i][w]$ as the maximum value achievable with the first i items and a weight limit w.
- Initialize:
  - $dp[0][w] = 0$ for all w (no items, no value)
  - $dp[i][0] = 0$ for all i (zero capacity, no value)
- Recursive relation:
  - If $w_i \leq w$:
    - $dp[i][w] = \max(dp[i-1][w], dp[i-1][w-w_i] + v_i)$
  - Otherwise:
    - $dp[i][w] = dp[i-1][w]$
- Construct the DP table iteratively.
- The answer will be $dp[n][W]$, where n is the number of items and W is the capacity of the knapsack.

0/1 Knapsack Problem: Dynamic Programming Visualization

- Explanation:
  - The rows represent items (1 to 5) with their weights in parentheses and the columns represent the weight capacity (0 to 10).
  - The value in each cell $dp[i][w]$ represents the maximum value achievable with the first i items and a weight limit w.
  - The final cell $dp[5][10]$ contains the maximum value achievable with all items and the full capacity.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0							
2,1	0							
4,3	0							
5,4	0							
3,2	0							

0/1 Knapsack Problem: Dynamic Programming Visualization

$$\text{dp[i][w]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max(v[i-1] + dp[i-1][w - w[i-1]], dp[i-1][w]) & \text{if } w[i-1] \leq w \\ dp[i-1][w] & \text{if } w[i-1] > w \end{cases}$$

$$dp[1][1] = 0,\ \text{because } w[1] > w$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
$v_i,\ w_i$	0	0	0	0	0	0	0	0
2,3	0	0						
2,1	0							
4,3	0							
5,4	0							
3,2	0							

0/1 Knapsack Problem: Dynamic Programming Visualization

$$\text{dp[i][w]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max(v[i-1] + dp[i-1][w - w[i-1]], dp[i-1][w]) & \text{if } w[i-1] \leq w \\ dp[i-1][w] & \text{if } w[i-1] > w \end{cases}$$

$$dp[1][2] = 0, \ \text{because } w[1] > w$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0	0	0					
2,1	0							
4,3	0							
5,4	0							
3,2	0							

0/1 Knapsack Problem: Dynamic Programming Visualization

$$\text{dp[i][w]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max(v[i-1] + dp[i-1][w - w[i-1]], dp[i-1][w]) & \text{if } w[i-1] \leq w \\ dp[i-1][w] & \text{if } w[i-1] > w \end{cases}$$

$$dp[1][3] = \max(v[0] + dp[0][3-3], dp[0][3]) \text{ since } w[0] \leq w$$

$$dp[1][3] = \max(2+0, 0) = 2$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0	0	0	2				
2,1	0							
4,3	0							
5,4	0							
3,2	0							

0/1 Knapsack Problem: Dynamic Programming Visualization

$$\text{dp[i][w]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max(v[i-1] + dp[i-1][w - w[i-1]], dp[i-1][w]) & \text{if } w[i-1] \leq w \\ dp[i-1][w] & \text{if } w[i-1] > w \end{cases}$$

$$dp[1][4] = \max(v[0] + dp[0][4-3], dp[0][4]) \text{ since } w[0] \leq w$$

$$dp[1][4] = \max(2 + 0, 0) = 2$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2			
2,1	0							
4,3	0							
5,4	0							
3,2	0							

0/1 Knapsack Problem: Dynamic Programming Visualization

$$\text{dp[i][w]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max(v[i-1] + dp[i-1][w - w[i-1]], dp[i-1][w]) & \text{if } w[i-1] \leq w \\ dp[i-1][w] & \text{if } w[i-1] > w \end{cases}$$

$$dp[2][1] = \max(v[1] + dp[1][1-1], dp[1][1]) \text{ since } w[1] \leq w$$

$$dp[2][1] = \max(2 + 0, 0) = 2$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0							
4,3	0							
5,4	0							
3,2	0							

0/1 Knapsack Problem: Dynamic Programming Visualization

$$\text{dp[i][w]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max(v[i-1] + dp[i-1][w - w[i-1]], dp[i-1][w]) & \text{if } w[i-1] \leq w \\ dp[i-1][w] & \text{if } w[i-1] > w \end{cases}$$

$$dp[2][2] = \max(v[1] + dp[1][2-1], dp[1][2]) \text{ since } w[1] \leq w$$

$$dp[2][2] = \max(2 + 0, 0) = 2$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2					
4,3	0							
5,4	0							
3,2	0							

0/1 Knapsack Problem: Dynamic Programming Visualization

$$\text{dp[i][w]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max(v[i-1] + dp[i-1][w - w[i-1]], dp[i-1][w]) & \text{if } w[i-1] \leq w \\ dp[i-1][w] & \text{if } w[i-1] > w \end{cases}$$

$$dp[2][3] = \max(v[1] + dp[1][3-1], dp[1][3]) \text{ since } w[1] \leq w$$

$$dp[2][3] = \max(2+2, 2) = 4$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2				
4,3	0							
5,4	0							
3,2	0							

0/1 Knapsack Problem: Dynamic Programming Visualization

$$\mathrm{dp[i][w]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max(v[i-1] + dp[i-1][w - w[i-1]], dp[i-1][w]) & \text{if } w[i-1] \leq w \\ dp[i-1][w] & \text{if } w[i-1] > w \end{cases}$$

$$dp[2][4] = \max(v[1] + dp[1][4-1], dp[1][4]) \text{ since } w[1] \leq w$$

$$dp[2][4] = \max(2 + 2, 2) = 4$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2	4			
4,3	0							
5,4	0							
3,2	0							

0/1 Knapsack Problem: Dynamic Programming Visualization

$$\text{dp[i][w]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max(v[i-1] + dp[i-1][w - w[i-1]], dp[i-1][w]) & \text{if } w[i-1] \leq w \\ dp[i-1][w] & \text{if } w[i-1] > w \end{cases}$$

$$dp[2][5] = \max(v[1] + dp[1][5-1], dp[1][5]) \text{ since } w[1] \leq w$$

$$dp[2][5] = \max(2+2, 2) = 4$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2	4	4		
4,3	0							
5,4	0							
3,2	0							

0/1 Knapsack Problem: Dynamic Programming Visualization

$$\text{dp[i][w]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max(v[i-1] + dp[i-1][w - w[i-1]], dp[i-1][w]) & \text{if } w[i-1] \leq w \\ dp[i-1][w] & \text{if } w[i-1] > w \end{cases}$$

$$dp[2][6] = \max(v[1] + dp[1][6-1], dp[1][6]) \text{ since } w[1] \leq w$$

$$dp[2][6] = \max(2 + 2, 2) = 4$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2	4	4	4	
4,3	0							
5,4	0							
3,2	0							

0/1 Knapsack Problem: Dynamic Programming Visualization

$$\text{dp[i][w]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max(v[i-1] + dp[i-1][w - w[i-1]], dp[i-1][w]) & \text{if } w[i-1] \leq w \\ dp[i-1][w] & \text{if } w[i-1] > w \end{cases}$$

$$dp[2][7] = \max(v[1] + dp[1][7-1], dp[1][7]) \text{ since } w[1] \leq w$$

$$dp[2][7] = \max(2+2, 2) = 4$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2	4	4	4	4
4,3	0							
5,4	0							
3,2	0							

0/1 Knapsack Problem: Dynamic Programming Visualization

$$\text{dp[i][w]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max(v[i-1] + dp[i-1][w - w[i-1]], dp[i-1][w]) & \text{if } w[i-1] \leq w \\ dp[i-1][w] & \text{if } w[i-1] > w \end{cases}$$

$$dp[3][1] = dp[2][1] \text{ since } w[2] > w$$

$$dp[3][1] = 2$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2	4	4	4	4
4,3	0	2						
5,4	0							
3,2	0							

0/1 Knapsack Problem: Dynamic Programming Visualization

$$\text{dp[i][w]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max(v[i-1] + dp[i-1][w - w[i-1]], dp[i-1][w]) & \text{if } w[i-1] \leq w \\ dp[i-1][w] & \text{if } w[i-1] > w \end{cases}$$

$$dp[3][2] = dp[2][2] \text{ since } w[2] > w$$

$$dp[3][2] = 2$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i , w_i	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2	4	4	4	4
4,3	0	2	2					
5,4	0							
3,2	0							

0/1 Knapsack Problem: Dynamic Programming Visualization

$$\text{dp[i][w]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max(v[i-1] + dp[i-1][w - w[i-1]], dp[i-1][w]) & \text{if } w[i-1] \leq w \\ dp[i-1][w] & \text{if } w[i-1] > w \end{cases}$$

$$dp[3][3] = \max(v[2] + dp[2][3-3], dp[2][3]) \text{ since } w[2] \leq w$$

$$dp[3][3] = \max(4 + 0, 4) = 4$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2	4	4	4	4
4,3	0	2	2	4				
5,4	0							
3,2	0							

0/1 Knapsack Problem: Dynamic Programming Visualization

$$\text{dp[i][w]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max(v[i-1] + dp[i-1][w - w[i-1]], dp[i-1][w]) & \text{if } w[i-1] \leq w \\ dp[i-1][w] & \text{if } w[i-1] > w \end{cases}$$

$$dp[3][4] = \max(v[2] + dp[2][4-3], dp[2][4]) \text{ since } w[2] \leq w$$

$$dp[3][4] = \max(4 + 0, 4) = 4$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
$v_i,\ w_i$	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2	4	4	4	4
4,3	0	2	2	4	6			
5,4	0							
3,2	0							

0/1 Knapsack Problem: Dynamic Programming Visualization

$$\text{dp[i][w]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max(v[i-1] + dp[i-1][w - w[i-1]], dp[i-1][w]) & \text{if } w[i-1] \leq w \\ dp[i-1][w] & \text{if } w[i-1] > w \end{cases}$$

$$dp[3][5] = \max(v[2] + dp[2][5-3], dp[2][5]) \text{ since } w[2] \leq w$$

$$dp[3][5] = \max(4 + 2, 4) = 6$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2	4	4	4	4
4,3	0	2	2	4	6	6		
5,4	0							
3,2	0							

0/1 Knapsack Problem: Dynamic Programming Visualization

$$\text{dp[i][w]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max(v[i-1] + dp[i-1][w - w[i-1]], dp[i-1][w]) & \text{if } w[i-1] \leq w \\ dp[i-1][w] & \text{if } w[i-1] > w \end{cases}$$

$$dp[3][6] = \max(v[2] + dp[2][6-3], dp[2][6]) \text{ since } w[2] \leq w$$

$$dp[3][6] = \max(4 + 2, 4) = 6$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2	4	4	4	4
4,3	0	2	2	4	6	6	6	
5,4	0							
3,2	0							

0/1 Knapsack Problem: Dynamic Programming Visualization

$$\text{dp[i][w]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max(v[i-1] + dp[i-1][w - w[i-1]], dp[i-1][w]) & \text{if } w[i-1] \leq w \\ dp[i-1][w] & \text{if } w[i-1] > w \end{cases}$$

$$dp[3][7] = \max(v[2] + dp[2][7-3], dp[2][7]) \text{ since } w[2] \leq w$$

$$dp[3][7] = \max(4 + 2, 4) = 6$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2	4	4	4	4
4,3	0	2	2	4	6	6	6	8
5,4	0							
3,2	0							

0/1 Knapsack Problem: Dynamic Programming Visualization

$$\text{dp[i][w]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max(v[i-1] + dp[i-1][w - w[i-1]], dp[i-1][w]) & \text{if } w[i-1] \leq w \\ dp[i-1][w] & \text{if } w[i-1] > w \end{cases}$$

$$dp[4][1] = dp[3][1] \text{ since } w[3] > w$$

$$dp[4][1] = 2$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2	4	4	4	4
4,3	0	2	2	4	6	6	6	8
5,4	0	2						
3,2	0							

0/1 Knapsack Problem: Dynamic Programming Visualization

$$\text{dp[i][w]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max(v[i-1] + dp[i-1][w - w[i-1]], dp[i-1][w]) & \text{if } w[i-1] \le w \\ dp[i-1][w] & \text{if } w[i-1] > w \end{cases}$$

$$dp[4][2] = dp[3][2] \text{ since } w[3] > w$$

$$dp[4][2] = 2$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2	4	4	4	4
4,3	0	2	2	4	6	6	6	8
5,4	0	2	2					
3,2	0							

0/1 Knapsack Problem: Dynamic Programming Visualization

$$\text{dp[i][w]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max(v[i-1] + dp[i-1][w - w[i-1]], dp[i-1][w]) & \text{if } w[i-1] \leq w \\ dp[i-1][w] & \text{if } w[i-1] > w \end{cases}$$

$$dp[4][3] = dp[3][3] \text{ since } w[3] > w$$

$$dp[4][3] = 4$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2	4	4	4	4
4,3	0	2	2	4	6	6	6	8
5,4	0	2	2	4				
3,2	0							

0/1 Knapsack Problem: Dynamic Programming Visualization

$$\text{dp[i][w]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max(v[i-1] + dp[i-1][w - w[i-1]], dp[i-1][w]) & \text{if } w[i-1] \leq w \\ dp[i-1][w] & \text{if } w[i-1] > w \end{cases}$$

$$dp[4][4] = \max(v[3] + dp[3][4-4], dp[3][4]) \text{ since } w[3] \leq w$$

$$dp[4][4] = \max(5 + 0, 4) = 5$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
$v_i,\ w_i$	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2	4	4	4	4
4,3	0	2	2	4	6	6	6	8
5,4	0	2	2	4	6			
3,2	0							

0/1 Knapsack Problem: Dynamic Programming Visualization

$$\text{dp[i][w]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max(v[i-1] + dp[i-1][w - w[i-1]], dp[i-1][w]) & \text{if } w[i-1] \leq w \\ dp[i-1][w] & \text{if } w[i-1] > w \end{cases}$$

$$dp[4][5] = \max(v[3] + dp[3][5-4], dp[3][5]) \text{ since } w[3] \leq w$$

$$dp[4][5] = \max(5 + 2, 6) = 7$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2	4	4	4	4
4,3	0	2	2	4	6	6	6	8
5,4	0	2	2	4	6	7		
3,2	0							

0/1 Knapsack Problem: Dynamic Programming Visualization

$$\text{dp[i][w]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max(v[i-1] + dp[i-1][w - w[i-1]], dp[i-1][w]) & \text{if } w[i-1] \leq w \\ dp[i-1][w] & \text{if } w[i-1] > w \end{cases}$$

$$dp[4][6] = \max(v[3] + dp[3][6-4], dp[3][6]) \text{ since } w[3] \leq w$$

$$dp[4][6] = \max(5 + 2, 6) = 7$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
$v_i\,,\ w_i$	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2	4	4	4	4
4,3	0	2	2	4	6	6	6	8
5,4	0	2	2	4	6	7	7	
3,2	0							

0/1 Knapsack Problem: Dynamic Programming Visualization

$$\text{dp[i][w]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max(v[i-1] + dp[i-1][w - w[i-1]], dp[i-1][w]) & \text{if } w[i-1] \leq w \\ dp[i-1][w] & \text{if } w[i-1] > w \end{cases}$$

$$dp[4][7] = \max(v[3] + dp[3][7-4], dp[3][7]) \text{ since } w[3] \leq w$$

$$dp[4][7] = \max(5 + 4, 6) = 9$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2	4	4	4	4
4,3	0	2	2	4	6	6	6	8
5,4	0	2	2	4	6	7	7	9
3,2	0							

0/1 Knapsack Problem: Dynamic Programming Visualization

$$\text{dp[i][w]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max(v[i-1] + dp[i-1][w - w[i-1]], dp[i-1][w]) & \text{if } w[i-1] \leq w \\ dp[i-1][w] & \text{if } w[i-1] > w \end{cases}$$

$$dp[5][1] = dp[4][1] \text{ since } w[4] > w$$

$$dp[5][1] = 2$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
$v_i,\ w_i$	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2	4	4	4	4
4,3	0	2	2	4	6	6	6	8
5,4	0	2	2	4	6	7	7	9
3,2	0	2						

0/1 Knapsack Problem: Dynamic Programming Visualization

$$\text{dp[i][w]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max(v[i-1] + dp[i-1][w - w[i-1]], dp[i-1][w]) & \text{if } w[i-1] \leq w \\ dp[i-1][w] & \text{if } w[i-1] > w \end{cases}$$

$$dp[5][2] = \max(v[4] + dp[4][2-2], dp[4][2]) \text{ since } w[4] \leq w$$

$$dp[5][2] = \max(3 + 0, 2) = 3$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
$v_i,\ w_i$	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2	4	4	4	4
4,3	0	2	2	4	6	6	6	8
5,4	0	2	2	4	6	7	7	9
3,2	0	2	3					

0/1 Knapsack Problem: Dynamic Programming Visualization

$$\text{dp[i][w]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max(v[i-1] + dp[i-1][w - w[i-1]], dp[i-1][w]) & \text{if } w[i-1] \le w \\ dp[i-1][w] & \text{if } w[i-1] > w \end{cases}$$

$$dp[5][3] = \max(v[4] + dp[4][3-2], dp[4][3]) \text{ since } w[4] \le w$$

$$dp[5][3] = \max(3+2, 4) = 5$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2	4	4	4	4
4,3	0	2	2	4	6	6	6	8
5,4	0	2	2	4	6	7	7	9
3,2	0	2	3	5				

0/1 Knapsack Problem: Dynamic Programming Visualization

$$\text{dp[i][w]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max(v[i-1] + dp[i-1][w - w[i-1]], dp[i-1][w]) & \text{if } w[i-1] \leq w \\ dp[i-1][w] & \text{if } w[i-1] > w \end{cases}$$

$$dp[5][4] = \max(v[4] + dp[4][4-2], dp[4][4]) \text{ since } w[4] \leq w$$

$$dp[5][4] = \max(3 + 2, 5) = 6$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2	4	4	4	4
4,3	0	2	2	4	6	6	6	8
5,4	0	2	2	4	6	7	7	9
3,2	0	2	3	5	6			

0/1 Knapsack Problem: Dynamic Programming Visualization

$$\text{dp[i][w]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max(v[i-1] + dp[i-1][w - w[i-1]], dp[i-1][w]) & \text{if } w[i-1] \leq w \\ dp[i-1][w] & \text{if } w[i-1] > w \end{cases}$$

$$dp[5][5] = \max(v[4] + dp[4][5-2], dp[4][5]) \text{ since } w[4] \leq w$$

$$dp[5][5] = \max(3 + 4, 7) = 7$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2	4	4	4	4
4,3	0	2	2	4	6	6	6	8
5,4	0	2	2	4	6	7	7	9
3,2	0	2	3	5	6	7		

0/1 Knapsack Problem: Dynamic Programming Visualization

$$\text{dp[i][w]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max(v[i-1] + dp[i-1][w - w[i-1]], dp[i-1][w]) & \text{if } w[i-1] \leq w \\ dp[i-1][w] & \text{if } w[i-1] > w \end{cases}$$

$$dp[5][6] = \max(v[4] + dp[4][6-2], dp[4][6]) \text{ since } w[4] \leq w$$

$$dp[5][6] = \max(3+4, 7) = 9$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2	4	4	4	4
4,3	0	2	2	4	6	6	6	8
5,4	0	2	2	4	6	7	7	9
3,2	0	2	3	5	6	7	9	

0/1 Knapsack Problem: Dynamic Programming Visualization

$$\text{dp[i][w]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max(v[i-1] + dp[i-1][w - w[i-1]], dp[i-1][w]) & \text{if } w[i-1] \le w \\ dp[i-1][w] & \text{if } w[i-1] > w \end{cases}$$

$$dp[5][7] = \max(v[4] + dp[4][7-2], dp[4][7]) \text{ since } w[4] \le w$$

$$dp[5][7] = \max(3 + 6, 9) = 10$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2	4	4	4	4
4,3	0	2	2	4	6	6	6	8
5,4	0	2	2	4	6	7	7	9
3,2	0	2	3	5	6	7	9	10

0/1 Knapsack Problem: Which items to take

- Highlight the final value (i.e, 10) in the table to identify the maximum value that can be obtained.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2	4	4	4	4
4,3	0	2	2	4	6	6	6	8
5,4	0	2	2	4	6	7	7	9
3,2	0	2	3	5	6	7	9	10

0/1 Knapsack Problem: Which items to take

- Highlight the final value (i.e, 10) in the table to identify the maximum value that can be obtained.
- Trace back to determine which items to include.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2	4	4	4	4
4,3	0	2	2	4	6	6	6	8
5,4	0	2	2	4	6	7	7	9
3,2	0	2	3	5	6	7	9	10

0/1 Knapsack Problem: Which items to take

- Highlight the final value (i.e, 10) in the table to identify the maximum value that can be obtained.
- Trace back to determine which items to include.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2	4	4	4	4
4,3	0	2	2	4	6	6	6	8
5,4	0	2	2	4	6	7	7	9
3,2	0	2	3	5	6	7	9	10

0/1 Knapsack Problem: Which items to take

- Highlight the final value (i.e, 10) in the table to identify the maximum value that can be obtained.
- Trace back to determine which items to include.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2	4	4	4	4
4,3	0	2	2	4	6	6	6	8
5,4	0	2	2	4	6	7	7	9
3,2	0	2	3	5	6	7	9	10

- If the value in the current cell is different from the value directly above it, it means the item corresponding to the current row was included.
- Item with value 3 and weight 2 was included (since 10 != 9)
- Remaining capacity: 7 - 2 = **5**

0/1 Knapsack Problem: Which items to take

- Highlight the final value (i.e, 10) in the table to identify the maximum value that can be obtained.
- Trace back to determine which items to include.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2	4	4	4	4
4,3	0	2	2	4	6	6	6	8
5,4	0	2	2	4	6	7	7	9
3,2	0	2	3	5	6	7	9	10

- Item with value 5 and weight 4 was included (since 7 != 6)
- Remaining capacity: 5 - 4 = **1**

0/1 Knapsack Problem: Which items to take

- Highlight the final value (i.e, 10) in the table to identify the maximum value that can be obtained.
- Trace back to determine which items to include.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2	4	4	4	4
4,3	0	2	2	4	6	6	6	8
5,4	0	2	2	4	6	7	7	9
3,2	0	2	3	5	6	7	9	10

- We don't include this item (v = 4, w =3) since 2 = 2
- We go up one row and check again

0/1 Knapsack Problem: Which items to take

- Highlight the final value (i.e, 10) in the table to identify the maximum value that can be obtained.
- Trace back to determine which items to include.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_i, w_i	0	0	0	0	0	0	0	0
2,3	0	0	0	2	2	2	2	2
2,1	0	2	2	2	4	4	4	4
4,3	0	2	2	4	6	6	6	8
5,4	0	2	2	4	6	7	7	9
3,2	0	2	3	5	6	7	9	10

- Item with value 2 and weight 1 was included (since 2 != 0)
- Remaining capacity: 1 - 1 = **0**
- We stop when we get to capacity of 0

0/1 Knapsack Problem: Runtime Analysis

- The dynamic programming solution for the 0/1 Knapsack Problem has a runtime complexity of $O(nW)$.
  - Here, n is the number of items and W is the maximum weight capacity of the knapsack.
- **Space Complexity:**
  - The space complexity is also $O(nW)$ because we need to store the values for each sub-problem in a table of size $n \times W$.
- **Why $O(nW)$?**
  - We iterate over each item (n iterations) and for each item, we iterate over the possible weight capacities from 0 to W.
  - For each sub-problem, we compute the value using the recurrence relation $dp[i][w] = \max(dp[i-1][w], dp[i-1][w-w_i] + v_i)$.

0/1 Knapsack Problem: Runtime Analysis

- **Comparison to Other Approaches:**
  - A brute force solution would examine all possible subsets of items, resulting in $O(2^n)$ time complexity.
  - The dynamic programming approach significantly reduces the runtime, making it feasible for larger instances with a time complexity of $O(nW)$.
- **Example Runtime:**
  - If $n = 5$ and $W = 7$, the DP table has $5 \times 8 = 40$ cells (considering weights from 0 to 7).
  - Filling each cell involves a constant time operation, leading to a total of $O(nW)$ operations.
- **Optimization:**
  - Space can be optimized to $O(W)$ by using a single-dimensional array and updating it in reverse order.
  - This technique can make the implementation less intuitive and harder to understand but is useful for saving space.

Longest Common Sub-sequence

Longest Common Sub-sequence: Definition

- The Longest Common Sub-sequence (LCS) problem is to find the longest sub-sequence common to two given sequences.
- A sub-sequence is a sequence derived by deleting some or no elements from the original sequence without changing the order of the remaining elements.
- The LCS problem has applications in bioinformatics, text comparison, and version control systems.

Longest Common Sub-sequence: Example

- Given two sequences X and Y, find the longest sub-sequence present in both.
- Example 1:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
  - $\text{LCS} = \{B, C, B, A\}$
- Example 2:
  - $X = \{A, G, G, T, A, B\}$
  - $Y = \{G, X, T, X, A, Y, B\}$
  - $\text{LCS} = \{G, T, A, B\}$

LCS: Recursive Solution

- Example:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, B\}$
- Recursive formula: $L("ABCBDAB", "BDCABB") = ?$
- Let say the sequences have lengths of n and m
- $L(n, m) = ?$
  - if elements $X[n] == Y[m] \Rightarrow L(n, m) = 1 + L(n-1, m-1)$
    - $L("ACDF", "CDEF") = 1 + L("ACDF", "CDEF")$
  - if elements $X[n] \neq Y[m] \Rightarrow$
    - $L(n, m) = \max\{L(n-1, m), L(n, m-1)\}$
    - $L("ACD", "CDE") = \max\{L("AC", "CDE"), L("ACD", "CD")\}$
  - Base case $L(n, m) = 0$, if $n = 0$ or $m = 0$

LCS: Recursive Solution

- For any i and j we can define $L(i,j)$ as the length of LCS of $X[1..i]$ and $Y[1..j]$.
- We can define the recursive formula:

$$L(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ L(i-1,j-1)+1 & \text{if } X_i = Y_j \\ \max(L(i-1,j), L(i,j-1)) & \text{otherwise} \end{cases}$$

- This recursive approach can be very slow due to overlapping subproblems and exponential time complexity.

LCS: DP Solution

- Compute $L(i,j)$ iteratively using a table:
  - Initialize a table L with dimensions $(m+1)\times$(n+1) where m is the length of X and n is the length of Y.
  - Fill the table using the following rules:

$$\text{for } i = 1 \text{ to } m:$$

$$\text{for } j = 1 \text{ to } n:$$

$$L(i,j) = \begin{cases} L(i-1,j-1)+1 & \text{if } X_i = Y_j \\ \max(L(i-1,j), L(i,j-1)) & \text{otherwise} \end{cases}$$

- Time complexity: $O(mn)$
- Space complexity: $O(mn)$

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""							
A							
B							
C							
B							
D							
A							
B							

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0						
B	0						
C	0						
B	0						
D	0						
A	0						
B	0						

LCS of any string with empty string is zero

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0					
B	0						
C	0						
B	0						
D	0						
A	0						
B	0						

$X[1] \neq Y[1] \Rightarrow A \neq B$
hence we take
$\max\{L(0,1), L(1,0)\} = max\{0, 0\}$

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0				
B	0						
C	0						
B	0						
D	0						
A	0						
B	0						

$X[1] \neq Y[2] \Rightarrow A \neq D$
hence we take
$\max\{L(0,2), L(1,1)\} =$
$max\{0, 0\}$

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0			
B	0						
C	0						
B	0						
D	0						
A	0						
B	0						

$X[1] \neq Y[3] \Rightarrow A \neq C$
hence we take
$\max\{L(0,3), L(1,2)\} =$
$max\{0,0\}$

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1		
B	0						
C	0						
B	0						
D	0						
A	0						
B	0						

$X[1] = Y[4]$ because $A = A$
hence we take
$1 + L(0, 3)\} = 1 + 0$

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	
B	0						
C	0						
B	0						
D	0						
A	0						
B	0						

$X[1] \neq Y[5] \Rightarrow A \neq B$
hence we take
$\max\{L(0,5), L(1,4)\} = max\{0, 1\}$

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0						
C	0						
B	0						
D	0						
A	0						
B	0						

$X[1] = Y[6]$ because $A = A$
hence we take
$1 + L(0, 5)\} = 1 + 0$

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1					
C	0						
B	0						
D	0						
A	0						
B	0						

$X[2] = Y[1]$ because $B = B$
hence we take
$1 + L(1, 0)\} = 1 + 0$

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1				
C	0						
B	0						
D	0						
A	0						
B	0						

$$X[2] \neq Y[2] \Rightarrow B \neq D$$

hence we take $\max\{L(1, 2), L(2, 1)\} = max\{0, 1\}$

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1			
C	0						
B	0						
D	0						
A	0						
B	0						

$$X[2] \neq Y[3] \Rightarrow B \neq C$$

hence we take $\max\{L(1,3), L(2,2)\} = max\{0, 1\}$

LCS: Example with DP Table

- Consider the sequences:
    - $X = \{A, B, C, B, D, A, B\}$
    - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1		
C	0						
B	0						
D	0						
A	0						
B	0						

$X[2] \neq Y[4] \Rightarrow B \neq A$
hence we take
$\max\{L(1,4), L(2,3)\} =$
$max\{0,1\}$

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	
C	0						
B	0						
D	0						
A	0						
B	0						

$X[2] = Y[5]$ because $B = B$
hence we take
$1 + L(1, 4)\} = 1 + 1$

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0						
B	0						
D	0						
A	0						
B	0						

$X[2] \neq Y[6] \Rightarrow B \neq A$
hence we take
$\max\{L(1,6), L(2,5)\} = max\{1,2\}$

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1					
B	0						
D	0						
A	0						
B	0						

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1				
B	0						
D	0						
A	0						
B	0						

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2			
B	0						
D	0						
A	0						
B	0						

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2		
B	0						
D	0						
A	0						
B	0						

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	
B	0						
D	0						
A	0						
B	0						

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0						
D	0						
A	0						
B	0						

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1					
D	0						
A	0						
B	0						

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1				
D	0						
A	0						
B	0						

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2			
D	0						
A	0						
B	0						

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2		
D	0						
A	0						
B	0						

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	
D	0						
A	0						
B	0						

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0						
A	0						
B	0						

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1					
A	0						
B	0						

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2				
A	0						
B	0						

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2			
A	0						
B	0						

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2		
A	0						
B	0						

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	
A	0						
B	0						

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0						
B	0						

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1					
B	0						

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2				
B	0						

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2			
B	0						

LCS: Example with DP Table

- Consider the sequences:
    - $X = \{A, B, C, B, D, A, B\}$
    - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3		
B	0						

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	
B	0						

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0						

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1					

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2				

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2			

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3		

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3	4	

LCS: Example with DP Table

- Consider the sequences:
  - $X = \{A, B, C, B, D, A, B\}$
  - $Y = \{B, D, C, A, B, A\}$
- The DP table for these sequences is as follows:

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3	4	4

LCS: Constructing the output

- The LCS itself can be constructed by backtracking through the table:
  - If the value of the current cell is greater than both the one above it and the one to its left
    - The characters in X and Y are the same, hence they are included in the result
    - Move (diagonally) one up and to the left and continue
  - If the value in the current cell is not greater than both
    - The characters in X and Y are different, hence are not included in the result
    - Move to the greater value cell of the two (above or to the left) and continue
  - We stop when we get to a cell with value of zero

LCS: Constructing the output

	""	B	D	C	A	B	A
""	**0**	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3	4	4

$LCS = \{ \quad \}$

LCS: Constructing the output

	""	B	D	C	A	B	A
""	**0**	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3	4	4

$LCS = \{ \quad \}$

LCS: Constructing the output

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	**0**	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3	4	4

$LCS = \{ \quad \text{A} \}$

LCS: Constructing the output

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	**0**	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3	4	4

$LCS = \{ \quad \text{A} \}$

LCS: Constructing the output

	""	B	D	C	A	B	A
""	**0**	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3	4	4

$LCS = \{ \quad \text{B, A} \}$

LCS: Constructing the output

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	**0**	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3	4	4

$LCS = \{\quad \text{B, A} \}$

LCS: Constructing the output

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	**0**	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3	4	4

$LCS = \{\ \text{C, B, A}\ \}$

LCS: Constructing the output

	""	B	D	C	A	B	A
""	**0**	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3	4	4

$LCS = \{$ C, B, A $\}$

LCS: Constructing the output

	""	**B**	**D**	**C**	**A**	**B**	**A**
""	**0**	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3	4	4

$LCS = \{$ B, C, B, A $\}$

LCS: Constructing the output

	""	B	D	C	A	B	A
""	**0**	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3	4	4

$LCS = \{$ B, C, B, A $\}$

LCS: Runtime Analysis

- The iterative solution for LCS has a time complexity of $O(mn)$,
  - Where m is the length of sequence X and n is the length of sequence Y.
- Space complexity is also $O(mn)$ due to the storage required for the DP table.
- This is significantly more efficient than the brute force or standard recursive approach, which have exponential time complexity.
- The iterative approach avoids redundant calculations by storing the results of sub-problems in a table.

Conclusion

Resource

- RNA Folding: Professor Bryce of Davidson College does a really good job at explaining Dynamic Programming over Intervals using RNA Secondary Structure as example

Conclusion

- Dynamic Programming is a powerful technique for solving optimization problems.
- Key steps: define sub-problems, find recursive relations, and solve iteratively.
- Common applications: interval scheduling, knapsack problem, sequence alignment, optimal BST, matrix chain multiplication, shortest paths.

Questions?