CP312
Algorithm Design and Analysis I

LECTURE 11: DYNAMIC PROGRAMMING

Dynamic Programming

- A technique for algorithm design to solve **optimization-type** problems
- Similar to **divide-and-conquer**:
  - Solves problems by combining solutions with smaller subproblems
- Unlike divide-and-conquer, dynamic programming is used when the **subproblems are not independent** and may **repeat**.
  - The DP solves each subproblem and then stores the result so that it can later reuse it without resolving the same subproblem.

Simple Example: Fibonacci Number

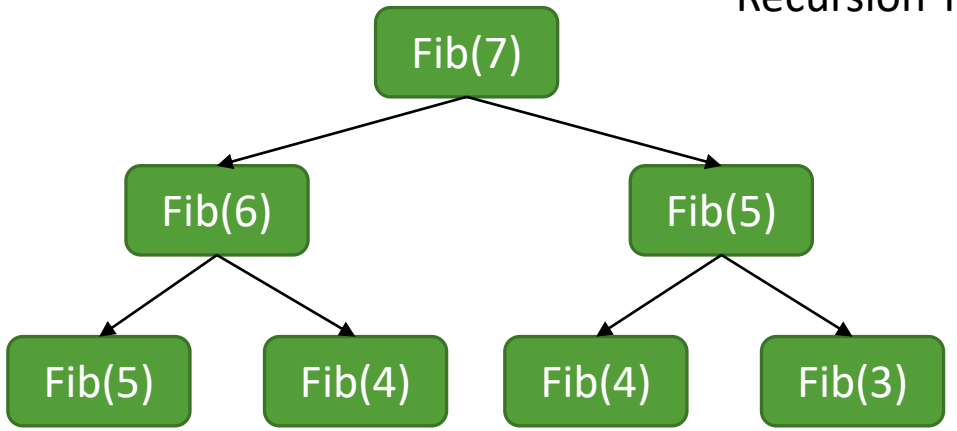
- Consider the algorithm for computing the Fibonacci number:

```
Fib(n):
If n ≤ 2 then
        return 1
Else
        return Fib(n − 1) + Fib(n − 2)
```

$$T(n) = T(n-1) + T(n-2) + \Theta(1)$$

Running Time is Exponential = $O(2^n)$

Recursion Tree



It keeps re-solving the same subproblem over and over again!

Simple Example: Fibonacci Number

- Consider an **alternative** way for computing the Fibonacci number:

MemFib(n):
Initialize array A of size n
If $n \leq 2$ then
    return 1
If $A[n]$ is not defined
    $A[n] = \mathbf{MemFib}(n-1) + \mathbf{MemFib}(n-2)$
return $A[n]$

Top-Down Memoization: Store the solutions of already solved subproblems then reuse the solutions when seen again.

Both have running time: $O(n)$

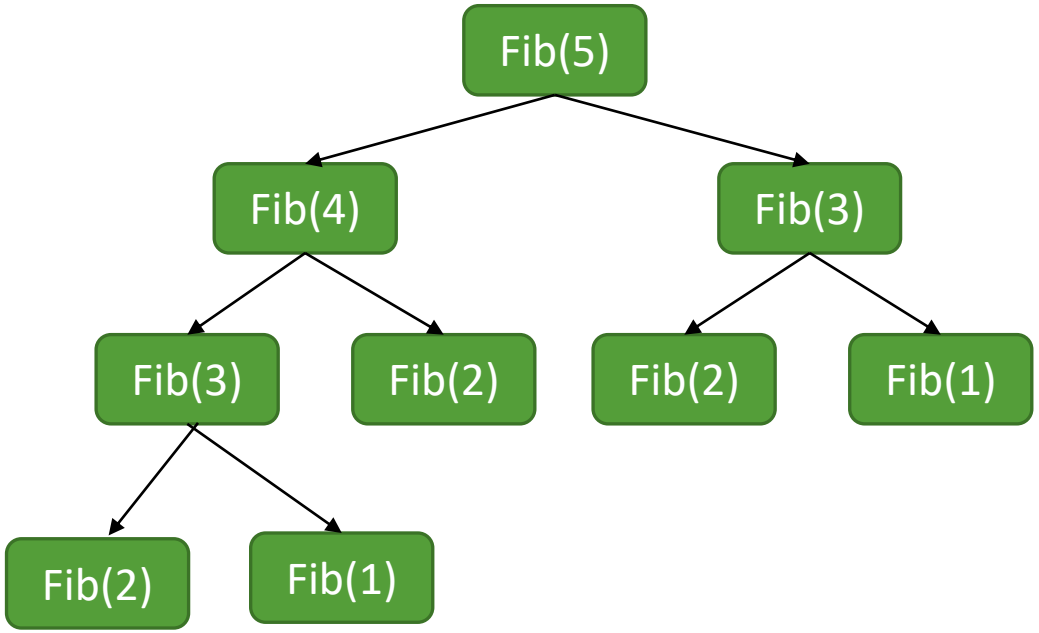
IterFib(n):
Initialize array A of size n
For $i = 1$ to n:
    If $i \leq 2$ then
        $A[i] = 1$
    If $A[i]$ is not defined
        $A[i] = A[i-1] + A[i-2]$
    return $A[i]$

Bottom-Up Iteration: Solve subproblems starting with the base case and moving towards larger subproblems.

This is **Dynamic Programming**. It can be thought of as "smart recursion"

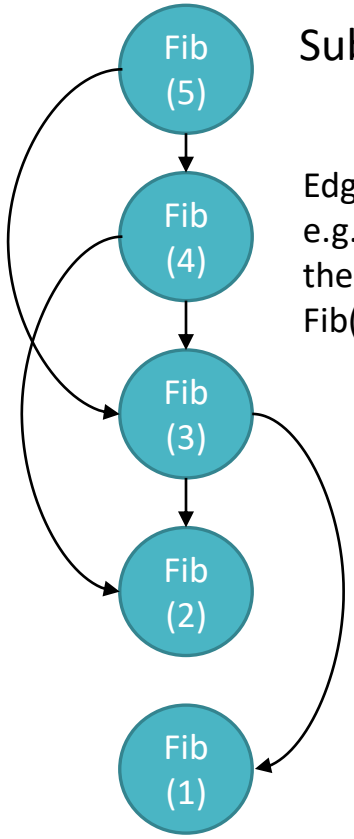
Recursion Trees and Subproblem Graphs

Recursion Tree



Models the execution of a **pure recursive** algorithm

Subproblem Graph



Edges denote dependencies: e.g. Subproblem Fib(5) needs the solution of subproblems Fib(4) and Fib(3)

Identifies the unique number of subproblems to build a **memorized DP** solution

Steps for Applying Dynamic Programming

1. Identify **optimal substructure**
   - An optimal solution to a problem contains optimal solutions to subproblems
2. Identify **overlapping subproblems**
   - A recursive solution contains a “small” number of distinct subproblems that are repeated many times.
3. Define a **recursive formulation**
   - Assuming solutions to sub-problems, recursively define the final solution
4. **Compute** the value of the final **optimal solution**
   - Store the solutions of sub-problems in a table and build the final solution out of the stored values (top-down or bottom-up)
   - A subproblem graph helps in identifying in what order to compute the values

Rod Cutting Problem

- **Problem:** Given a rod of length n and a table of prices for each length of rod, determine the maximum revenue to obtain by cutting up the rod and selling the pieces.
- **Input:** Initial length n and list of prices p_i for $i = 1, \ldots, n$
- **Output:** The maximum revenue resulting from any rod cutting

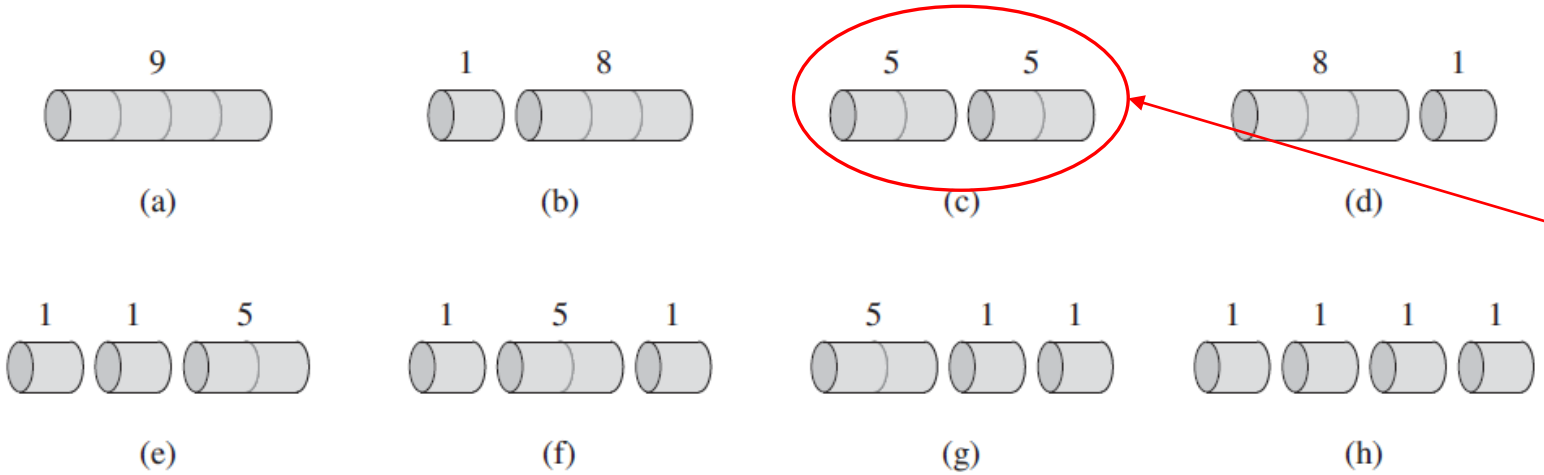
Rod Cutting Problem

- Example: Given the following rod and list of prices:

$n = 4$

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

- What are the possible ways of cutting this rod and what is the revenue for each way?



The rod cutting that will yield the highest revenue: **10**

Rod Cutting Problem

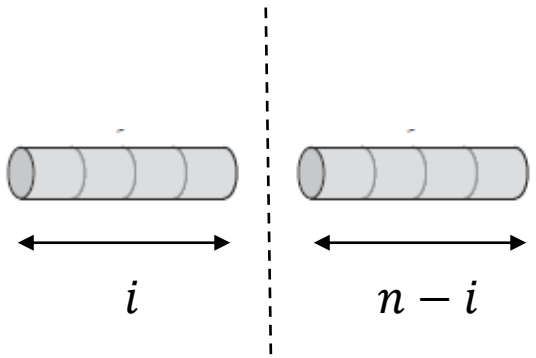
- Naïve Solution:
- We can cut up a rod of length n in 2^{n-1} different ways, since we have an independent option of cutting, or not cutting, at distance i from the left end.
- Running time: $O(2^n)$ Exponential!

Steps for DP: Rod Cutting

1. Identify **optimal substructure**
   - An optimal solution to a problem contains optimal solutions to subproblems
2. Identify **overlapping subproblems**
   - A recursive solution contains a “small” number of distinct subproblems that are repeated many times.
3. Define a **recursive formulation**
   - Assuming solutions to sub-problems, recursively define the final solution
4. Compute the value of the final optimal solution
   - Store the solutions of sub-problems in a table and build the final solution out of the stored values (top-down or bottom-up)
   - A subproblem graph helps in identifying in what order to compute the values

Rod Cutting Problem

- We can come up with a DP algorithm based on the following observations:
  - The optimal solution contains a left piece at some length i and then a remaining part of length $n - i$
  - We only then have to consider how to divide the right part.



- We have optimal substructure (proof omitted)
- And we have overlapping subproblems

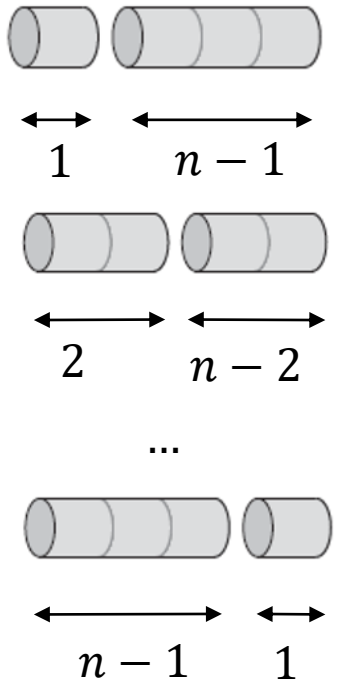
Rod Cutting: Recursive Formulation

- Let $R(n)$ denote the revenue we obtain by first cutting the rod at position i and then recursively cut the right part of length $n - i$

$$R(n) = p_i + R(n - i)$$

Profit from left part

Profit from remaining part



- How can we choose among all possibilities for the **first** initial cut?
- We try them all and pick the best:

$$R(n) = \max_{1 \le i \le n} \{p_i + R(n - i)\}$$

$$R(n) = \max_{1 \le i \le n} \{p_i + R(n-i)\}$$

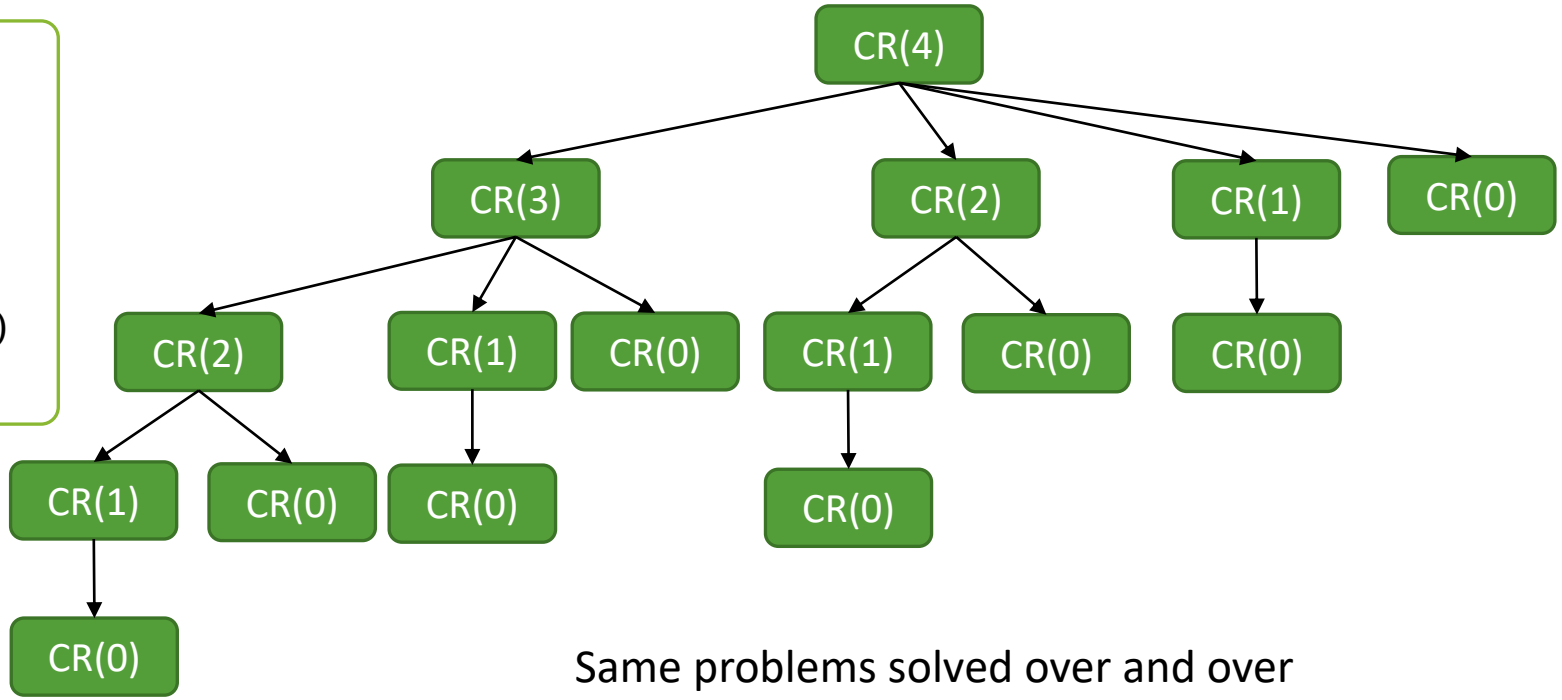
Rod Cutting: Recursive Formulation

- Straightforward Recursion

Cut-Rod(p, n):
If $n == 0$ then
    return 0
q $= -\infty$
For $i = 1$ to n
    $q = \max(q, p[i] + \textbf{Cut-Rod}(p, n-i))$
return q

How to speed this up?

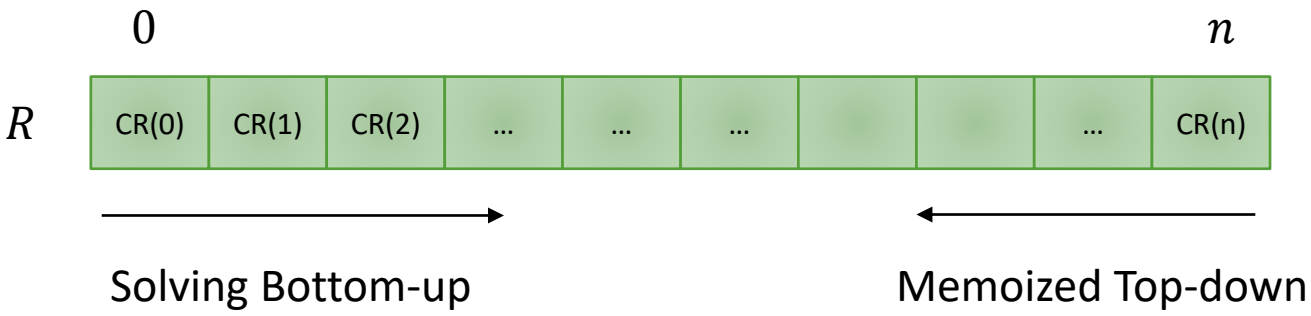
Recursion Tree

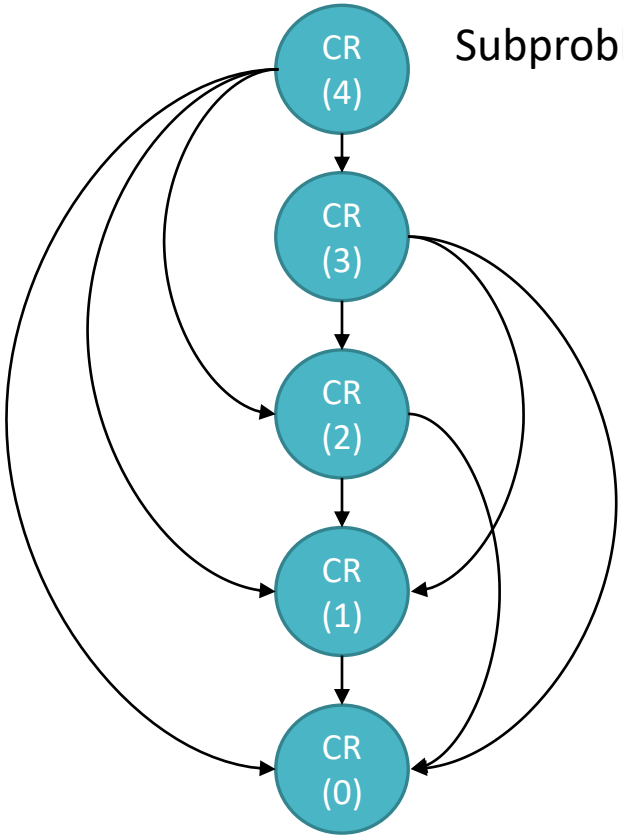


Same problems solved over and over again. Exponential time!

Rod Cutting: Memoized DP

- From the subproblem graph, we can see:
  - The dependencies between subproblems. For example, we need the solution to CR(2), CR(1), and CR(0) to solve CR(3).
  - The number of subproblems is $n + 1$ for a problem (rod) size n
  - The amount of memory to set up for memoization will be equal to the number of subproblems.





Rod Cutting: Memoized DP

- Use an array $r[0, \ldots, n]$ of solutions, initialized with -1

```
Mem-Cut-Rod(p, n, r):
If r[n] ≥ 0 then
        return r[n]
If n == 0
        q = 0
Else
        q = −∞
        for i = 1 to n
                q = max(q, p[i] + Mem-Cut-Rod(p, n − i, r))
r[n] = q
return q
```

R

Recursive:
Memoized Top-down

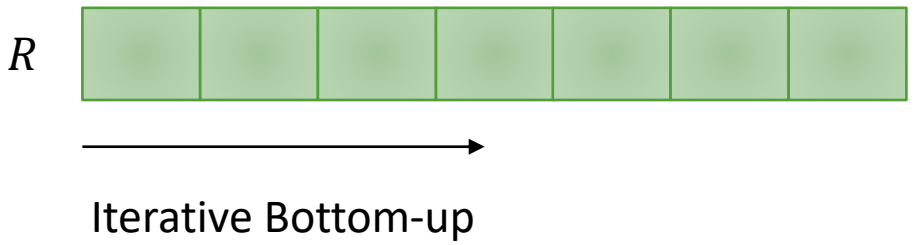
$$\max_{1 \le i \le n} \{p_i + R(n-i)\}$$

Rod Cutting: Bottom-up DP

- Use an array $r[0, \ldots, n]$ of solutions, initialized with -1

```
BottomUp-Cut-Rod(p, n, r):
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j
        q = max(q, p[i] + r[j − i])
    r[j] = q
return r[n]
```

$$\max_{1 \le i \le n} \{p_i + R(n-i)\}$$



Longest Common Subsequence (LCS)

- **Problem:** Given two sequences, find a longest (not necessarily contiguous) subsequence that is common in both sequences.
- **Input:** Two arrays $x[1, \dots m]$ and $y[1, \dots, n]$
- **Output:** A longest common subsequence in x and y

- Example:

$$x = [A, B, C, B, D, A, B]$$
$$y = [B, D, C, A, B, A]$$
$$LCS(x, y): (\boldsymbol{B}, \boldsymbol{C}, \boldsymbol{B}, \boldsymbol{A})$$

Longest Common Subsequence (LCS)

- Naïve solution: Brute-force
  - Check every subsequence $x[1, \dots, m]$ to see if it is also a subsequence of $y[1, \dots, n]$

- Analysis:
  - There are $\mathbf{2^m}$ subsequences of x (each bit-vector of length m determines a distinct subsequence of x)
  - For each subsequence of x, we make $O(n)$ comparisons to check if it is also in y
  - Worst-case running-time $= \boldsymbol{O(n2^m)}$

Exponential Time!

Can we do better?

Steps for Applying Dynamic Programming

1. Identify optimal substructure
   - An optimal solution to a problem contains optimal solutions to subproblems
2. Identify overlapping subproblems
   - A recursive solution contains a “small” number of distinct subproblems that are repeated many times.
3. Define a recursive formulation
   - Assuming solutions to sub-problems, recursively define the final solution
4. Compute the value of the final optimal solution
   - Store the solutions of sub-problems in a table and build the final solution out of the stored values (top-down or bottom-up)
   - A subproblem graph helps in identifying in what order to compute the values

Longest Common Subsequence (LCS)

- Simplification:

1. Look at the **length** of a longest-common subsequence
2. Extend the algorithm to find the LCS itself

- Notation: denote the length of a sequence s by $|s|$

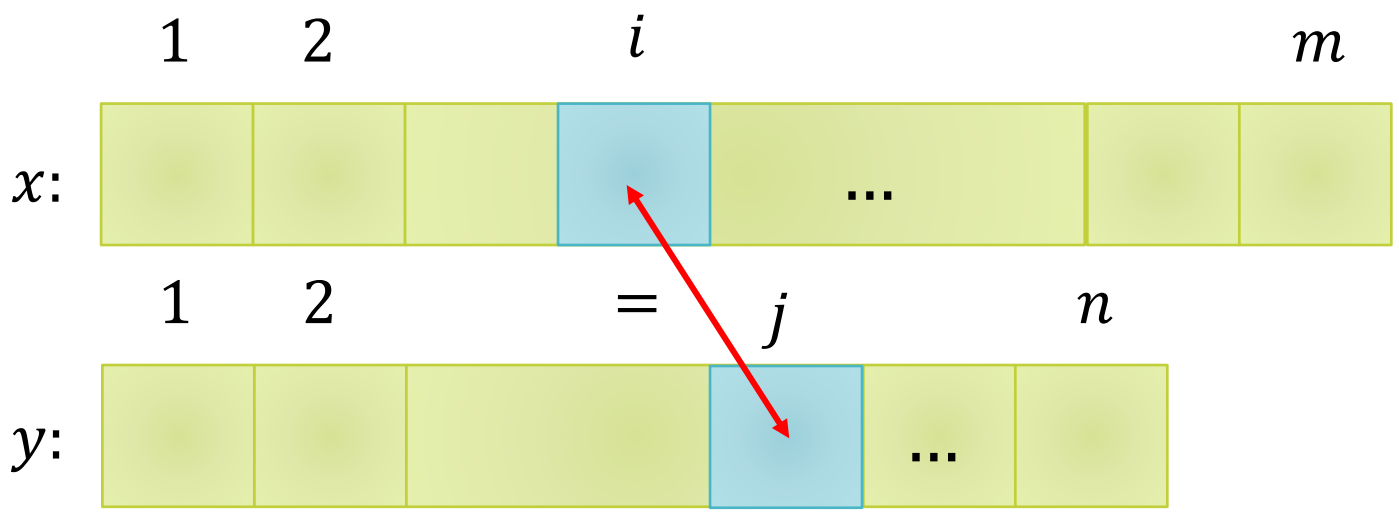
- Strategy: Consider **prefixes** of x and y
  - Define $c[i,j] = |LCS(x[1,\dots,i], y[1,\dots,j])|$
  - Then $c[m,n] = |LCS(x,y)|$

Longest Common Subsequence (LCS)

- Theorem:

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i] = y[j] \\ \max\{c[i-1,j], c[i,j-1]\} & \text{otherwise} \end{cases}$$

- Proof: Case $x[i] = y[j]$



$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i] = y[j] \\ \max\{c[i-1,j], c[i,j-1]\} & \text{otherwise} \end{cases}$$

Longest Common Subsequence (LCS)

- Proof (continued) for Case $x[i] = y[j]$

1. Let $z[1,\dots,k] = LCS(x[1,\dots,i], y[1,\dots,j])$ where $c[i,j] = k$.
   - Then $z[k] = x[i] = y[j]$
2. Thus, $z[1,\dots,k-1]$ is a common subsequence of $x[1,\dots,i-1]$ and $y[1,\dots,j-1]$
3. Furthermore, $z[1,\dots,k-1] = LCS(x[1,\dots,i-1], y[1,\dots,j-1])$
   - If it is not, then this means there exists a longer common subsequence w of $x[1,\dots,i-1]$ and $y[1,\dots,j-1]$ such that $|w| > k-1$ whereby appending x[i] or y[j] to w makes $|w|+1 > k$ (Contradiction!)
4. Therefore, $c[i-1,j-1] = k-1$, which implies that:

$$c[i,j] = c[i-1,j-1]+1$$

Longest Common Subsequence (LCS)

Property 1

<u>Optimal Substructure</u>

An optimal solution to a problem contains optimal solutions to subproblems

If $z = LCS(x, y)$ then any prefix of z is an LCS of a prefix of x and a prefix of y

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i] = y[j] \\ \max\{c[i-1,j], c[i,j-1]\} & \text{otherwise} \end{cases}$$

Recursive algorithm for LCS

LCS(x, y, i, j):

if $x[i] == y[j]$

$c[i,j] = \mathrm{LCS}(x, y, i-1, j-1) + 1$

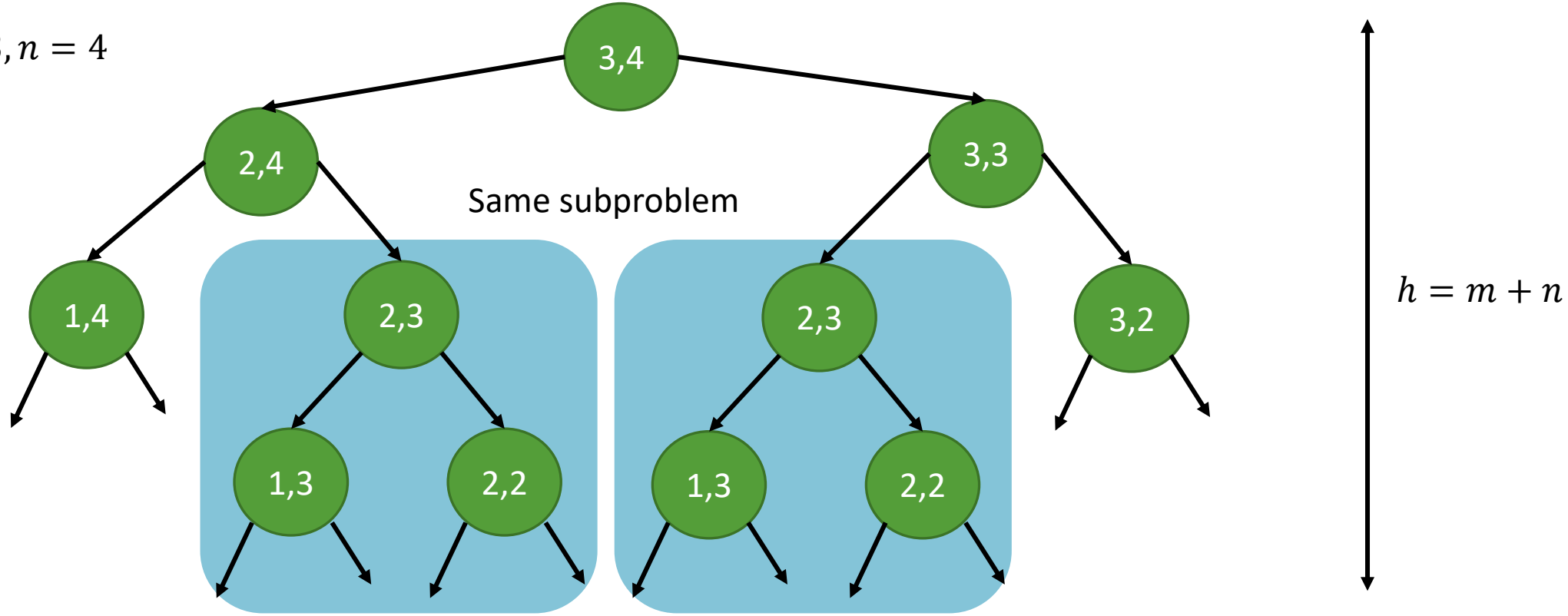
else $c[i,j] = \max\{\mathrm{LCS}(x, y, i-1, j), \mathrm{LCS}(x, y, i, j-1)\}$

- **Worst-case:** $x[i] \neq y[j]$ in which case the algorithm evaluates two subproblems, each with only one parameter decremented.

Recursion Tree for LCS

Height $= m + n \Rightarrow$ work potentially exponential

$m = 3, n = 4$



But we're solving subproblems already solved!

Longest Common Subsequence (LCS)

Property 2

<u>Overlapping Subproblems</u>

A recursive solution contains a “small” number of distinct subproblems that are repeated many times

The number of distinct LCS subproblems for two strings of lengths m and n is only mn

Memoization algorithm for LCS

- **Memoization**: After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

LCS(x, y, i, j):

if $c[i, j] = NULL$

if $x[i] == y[j]$

$c[i, j] = \text{LCS}(x, y, i-1, j-1) + 1$

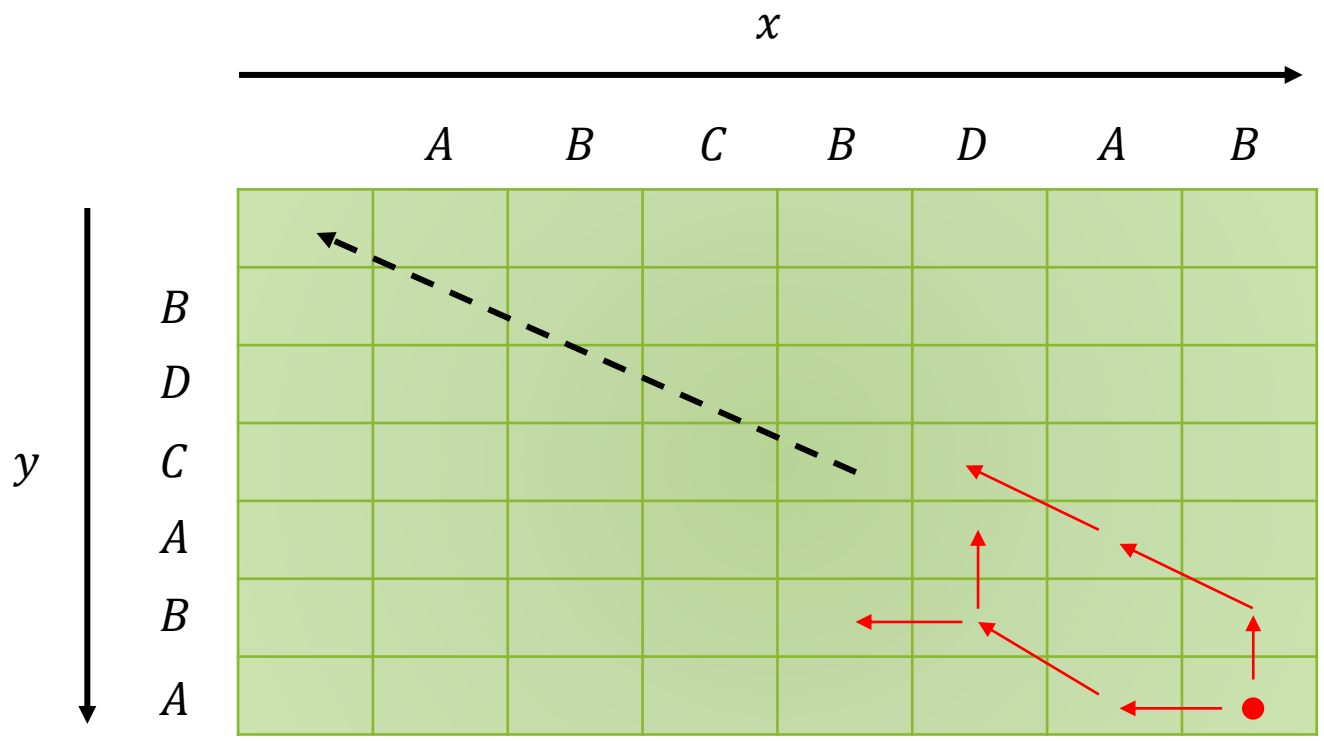
else $c[i, j] = \max\{\text{LCS}(x, y, i-1, j), \text{LCS}(x, y, i, j-1)\}$

Same as before

- Time = $\Theta(mn)$ = constant work per table entry
- Space = $\Theta(mn)$

Memoization algorithm for LCS

- The values in the table are computed **top-down**
- Time $= \Theta(mn)$

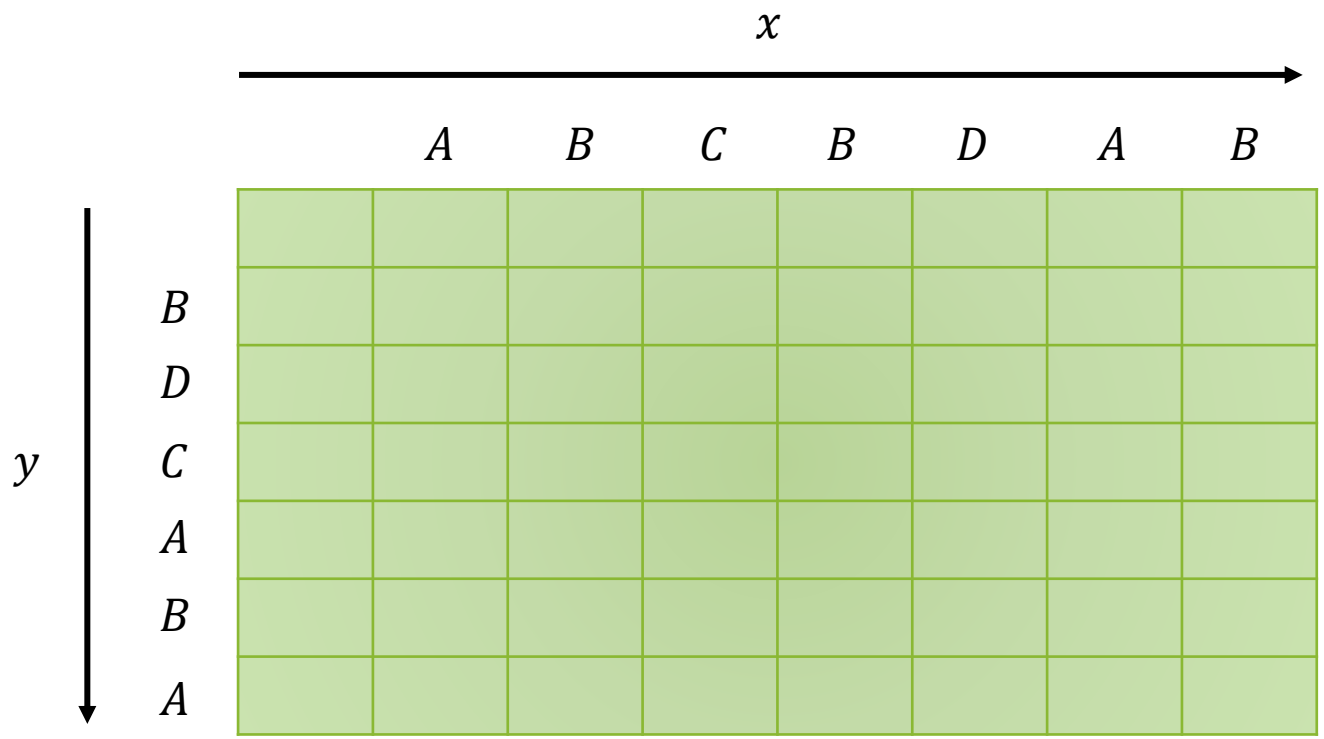


LCS(x, y, i, j):
  if $c[i, j] = NULL$
    if $x[i] == y[j]$
      $c[i, j] = \text{LCS}(x, y, i-1, j-1) + 1$
    else $c[i, j] = \max\{\text{LCS}(x, y, i-1, j), \text{LCS}(x, y, i, j-1)\}$

Start by running LCS(x, y, m, n)

Memoization algorithm for LCS

- The values in the table are computed **top-down**
- Time $= \Theta(mn)$



LCS(x, y, i, j):
  if $c[i, j] = NULL$
    if $x[i] == y[j]$
      $c[i, j] = \text{LCS}(x, y, i-1, j-1) + 1$
    else $c[i, j] = \max\{\text{LCS}(x, y, i-1, j), \text{LCS}(x, y, i, j-1)\}$

Memoization algorithm for LCS

- The values in the table are computed **top-down**
- Time $= \Theta(mn)$

x →

y ↓

		A	B	C	B	D	A	B
	0	0	0	0				
B	0	0	1	1				
D	0	0	1	1				
C	0	0	1	2				
A	0	1	1			…		
B	0	1	2					
A	0	1	2					4

LCS(x, y, i, j):
  if $c[i, j] = NULL$
    if $x[i] == y[j]$
      $c[i, j] = \text{LCS}(x, y, i-1, j-1) + 1$
    else $c[i, j] = \max\{\text{LCS}(x, y, i-1, j), \text{LCS}(x, y, i, j-1)\}$

Bottom-up algorithm for LCS

- Alternatively: Compute the table **bottom-up**
- Time $= \Theta(mn)$

x →

y ↓

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0							
D	0							
C	0							
A	0							
B	0							
A	0							

Bottom-up algorithm for LCS

- Alternatively: Compute the table **bottom-up**
- Time $= \Theta(mn)$

x →

y ↓

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

Reconstructing the LCS

- Reconstruct LCS by tracing backwards

x →

y ↓

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

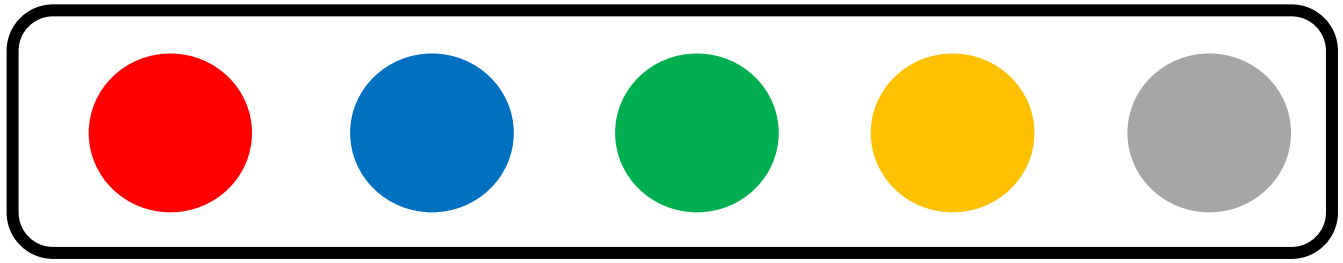
0/1 Knapsack Problem

- **Problem:** Given a set of items, find a combination of items to add to a bag of some capacity that yields the most value.
- **Input:** A set of n weight/value pairs $\mathrm{S} = \{(1, v_1, w_1), (2, v_2, w_2), \dots, (n, v_n, w_n)\}$ and capacity W
- **Output:** A subset $T \subseteq \{1,2, \dots, n\}$ that:

$$\text{Maximizes } \sum_{i \in T} v_i$$

$$\text{Subject to } \sum_{i \in T} w_i \leq W$$

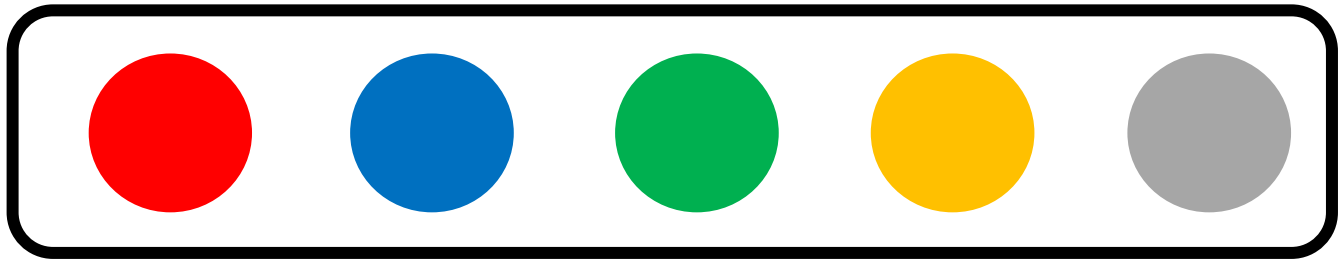
The Knapsack Problem



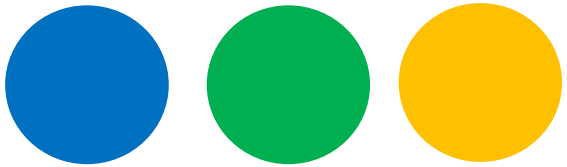
Weight (kg)	6	2	4	3	11
Value ($)	20	8	14	13	35

We have a knapsack that can only carry so much weight
Capacity: 10 kg

The Knapsack Problem



Weight (kg)	6	2	4	3	11
Value ($)	20	8	14	13	35



Total weight: 9
Total value: 35

Capacity: 10 kg

Suppose I only have one copy of each item
What is the most valuable way to fill the bag?

0/1 Knapsack Problem

- **Naïve solution:**
- Brute force: Try all possible subsets of T
  - E.g. Try $\{1\}$ then $\{1,2\}$ then $\{1,2,3\}$ then $\{1,3\}$ etc.
- Running time?
  - $\Theta(2^n)$ Not good!

Steps for DP: 0/1 Knapsack Problem

1. Identify **optimal substructure**
   - An optimal solution to a problem contains optimal solutions to subproblems
2. Identify **overlapping subproblems**
   - A recursive solution contains a “small” number of distinct subproblems that are repeated many times.
3. Define a **recursive formulation**
   - Assuming solutions to sub-problems, recursively define the final solution
4. Compute the value of the final optimal solution
   - Store the solutions of sub-problems in a table and build the final solution out of the stored values (top-down or bottom-up)
   - A subproblem graph helps in identifying in what order to compute the values

0/1 Knapsack Problem

1. Identify optimal substructure with overlapping subproblems.

First solve the problem for small knapsacks

Then larger knapsacks

Then larger knapsacks

0/1 Knapsack Problem

1. Identify optimal substructure with overlapping subproblems.

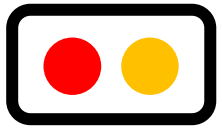
First solve the problem for small knapsacks

Then larger knapsacks

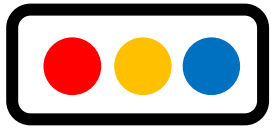
Then larger knapsacks

First solve the problem for few items

Then more items



Then more items



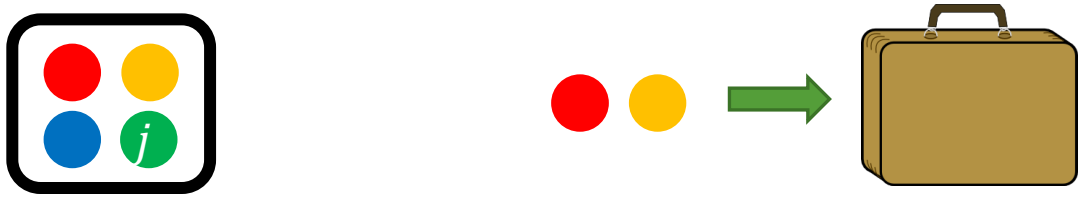
We need a two-dimensional table

0/1 Knapsack Problem

1. Identify optimal substructure with overlapping subproblems.

Case 1: If the optimal solution for j items <u>does not use</u> item j

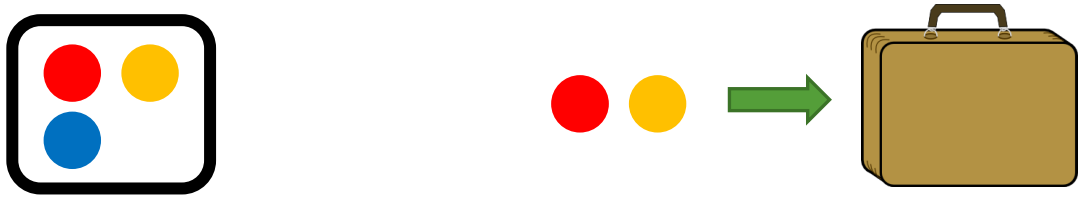
The first j items



Capacity: x
Value: V

Then the following is an optimal solution for $j - 1$ items:

The first $j - 1$ items



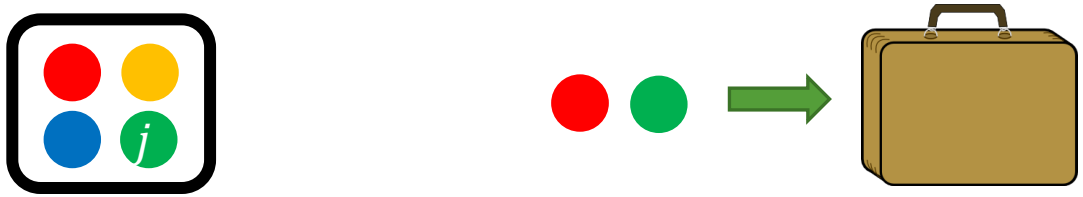
Capacity: x
Value: V

0/1 Knapsack Problem

1. Identify optimal substructure with overlapping subproblems.

Case 2: If the optimal solution for j items <u>uses</u> item j

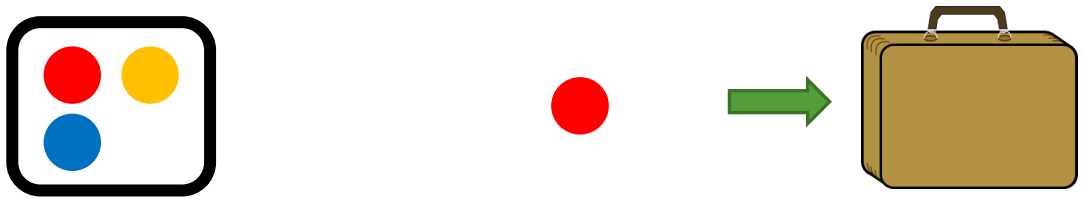
The first j items



Capacity: x
Value: V

Then the following is an optimal solution for $j - 1$ items:

The first $j - 1$ items



Capacity: $x - w_j$
Value: $V - v_j$

Steps for DP: 0/1 Knapsack Problem

1. Identify **optimal substructure** ✓
   - An optimal solution to a problem contains optimal solutions to subproblems
2. Identify **overlapping subproblems** ✓
   - A recursive solution contains a “small” number of distinct subproblems that are repeated many times.
3. Define a **recursive formulation**
   - Assuming solutions to sub-problems, recursively define the final solution
4. Compute the value of the final optimal solution
   - Store the solutions of sub-problems in a table and build the final solution out of the stored values (top-down or bottom-up)
   - A subproblem graph helps in identifying in what order to compute the values

0/1 Knapsack Problem

3. Define a recursive formulation.

Let $K[j, x]$ be the optimal value for capacity x with j items

$$K[j,x] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[j-1,x], K[j-1, x-w_j] + v_j\} & \text{otherwise} \end{cases}$$

Item j is not used

Item j is used

Steps for DP: 0/1 Knapsack Problem

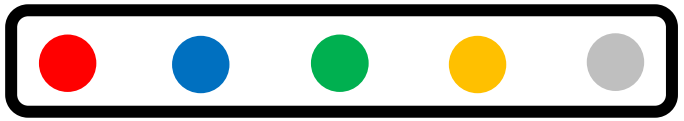
1. Identify **optimal substructure** ✓
   - An optimal solution to a problem contains optimal solutions to subproblems
2. Identify **overlapping subproblems** ✓
   - A recursive solution contains a “small” number of distinct subproblems that are repeated many times.
3. Define a **recursive formulation** ✓
   - Assuming solutions to sub-problems, recursively define the final solution
4. Compute the value of the final optimal solution
   - Store the solutions of sub-problems in a table and build the final solution out of the stored values (top-down or bottom-up)
   - A subproblem graph helps in identifying in what order to compute the values

$$K[j,x] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[j-1,x], K[j-1, x-w_j] + v_j\} & \text{otherwise} \end{cases}$$

0/1 Knapsack Problem

4. Compute the value of the optimal solution bottom-up

Knapsack$(W, \text{weights}, \text{values})$:
$n = \text{weights.length}$
for $x = 0$ **to** W and $j = 0$ **to** n
    $K[j,x] = 0$
for $x = 1$ **to** W
    for $j = 1$ **to** n
        $K[j,x] = K[j-1,x]$
        $w_j = \text{weights}[j]$
        $v_j = \text{values}[j]$
        if $w_j \leq x$
            $K[j,x] = \max\{K[j-1,x], K[j-1, x-w_j] + v_j\}$
return $K[n,W]$



weights	6	2	4	3	11
values	20	8	14	13	35

Runtime: $O(nW)$

Steps for DP: 0/1 Knapsack Problem

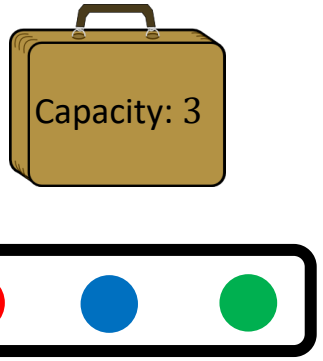
1. Identify **optimal substructure** ✓
   - An optimal solution to a problem contains optimal solutions to subproblems
2. Identify **overlapping subproblems** ✓
   - A recursive solution contains a “small” number of distinct subproblems that are repeated many times.
3. Define a **recursive formulation** ✓
   - Assuming solutions to sub-problems, recursively define the final solution
4. Compute the value of the final optimal solution ✓
   - Store the solutions of sub-problems in a table and build the final solution out of the stored values (top-down or **bottom-up**)
   - A subproblem graph helps in identifying in what order to compute the values

0/1 Knapsack Problem

- **Example:**

$K[j,x]$

j \ x	0	1	2	3
0	0	0	0	0
1	0			
2	0			
3	0			



weights	1	2	3
values	1	4	6

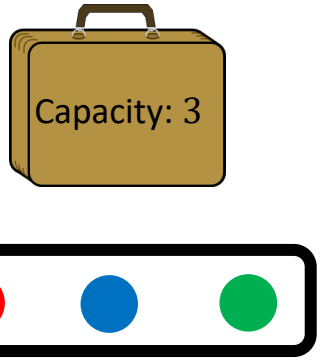
$$K[j,x] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[j-1,x], K[j-1, x-w_j] + v_j\} & \text{otherwise} \end{cases}$$

0/1 Knapsack Problem

- **Example:**

$K[j,x]$

j \ x	0	1	2	3
0	0	0	0	0
1	0	1		
2	0			
3	0			



weights	1	2	3
values	1	4	6

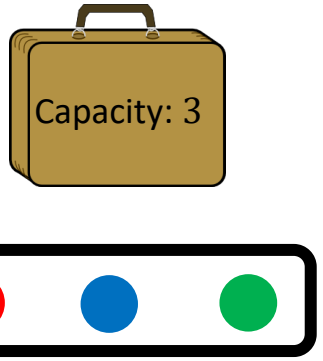
$$K[j,x] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[j-1,x], K[j-1, x-w_j] + v_j\} & \text{otherwise} \end{cases}$$

0/1 Knapsack Problem

- **Example:**

$K[j,x]$

j \ x	0	1	2	3
0	0	0	0	0
1	0	1		
2	0	1		
3	0			



weights	1	2	3
values	1	4	6

$$K[j,x] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[j-1,x], K[j-1, x-w_j] + v_j\} & \text{otherwise} \end{cases}$$

0/1 Knapsack Problem

- **Example:**

$K[j,x]$

j \ x	0	1	2	3
0	0	0	0	0
1	0	1		
2	0	1		
3	0	1		

weights	1	2	3
values	1	4	6

$$K[j,x] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[j-1,x], K[j-1, x-w_j] + v_j\} & \text{otherwise} \end{cases}$$

0/1 Knapsack Problem

- **Example:**

$K[j,x]$

j \ x	0	1	2	3
0	0	0	0	0
1	0	1	1	
2	0	1		
3	0	1		

weights	1	2	3
values	1	4	6

$$K[j,x] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[j-1,x], K[j-1, x-w_j] + v_j\} & \text{otherwise} \end{cases}$$

0/1 Knapsack Problem

- **Example:**

$K[j,x]$

j \ x	0	1	2	3
0	0	0	0	0
1	0	1	1	
2	0	1	4	
3	0	1		

weights	1	2	3
values	1	4	6

$$K[j,x] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[j-1,x], K[j-1, x-w_j] + v_j\} & \text{otherwise} \end{cases}$$

0/1 Knapsack Problem

- **Example:**

$K[j,x]$

j \ x	0	1	2	3
0	0	0	0	0
1	0	1	1	
2	0	1	4	
3	0	1	4	

weights	1	2	3
values	1	4	6

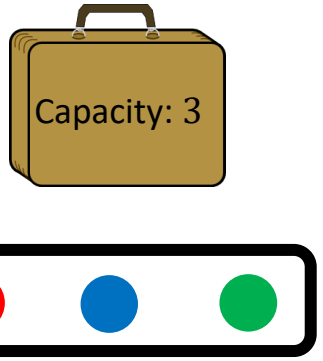
$$K[j,x] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[j-1,x], K[j-1, x-w_j] + v_j\} & \text{otherwise} \end{cases}$$

0/1 Knapsack Problem

- **Example:**

$K[j,x]$

j \ x	0	1	2	3
0	0	0	0	0
1	0	1	1	1
2	0	1	4	
3	0	1	4	



weights	1	2	3
values	1	4	6

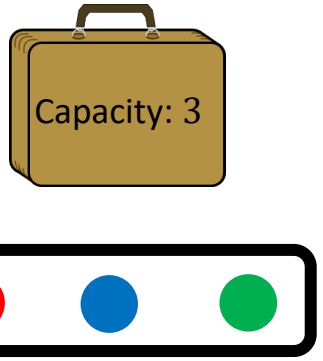
$$K[j,x] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[j-1,x], K[j-1, x-w_j] + v_j\} & \text{otherwise} \end{cases}$$

0/1 Knapsack Problem

- **Example:**

$K[j,x]$

j \ x	0	1	2	3
0	0	0	0	0
1	0	1	1	1
2	0	1	4	5
3	0	1	4	



weights	1	2	3
values	1	4	6

$$K[j,x] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[j-1,x], K[j-1, x-w_j] + v_j\} & \text{otherwise} \end{cases}$$

0/1 Knapsack Problem

- **Example:**

$K[j,x]$

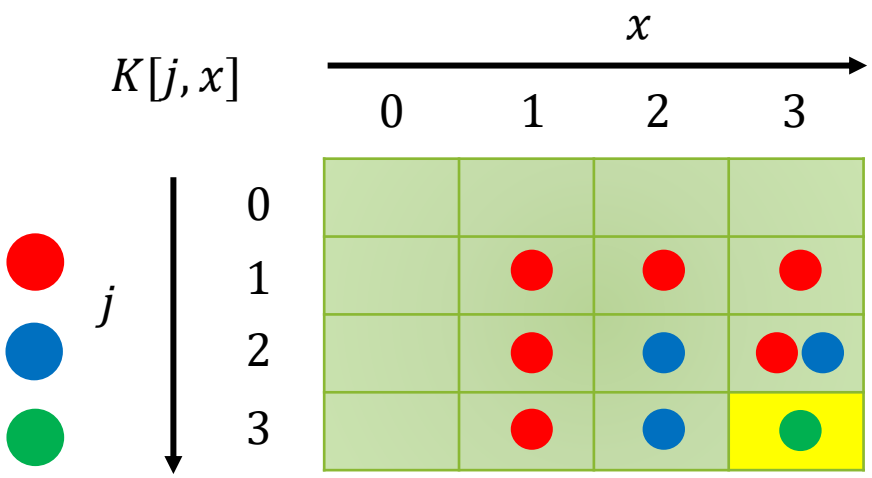
j \ x	0	1	2	3
0	0	0	0	0
1	0	1	1	1
2	0	1	4	5
3	0	1	4	6

weights	1	2	3
values	1	4	6

$$K[j,x] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[j-1,x], K[j-1, x-w_j] + v_j\} & \text{otherwise} \end{cases}$$

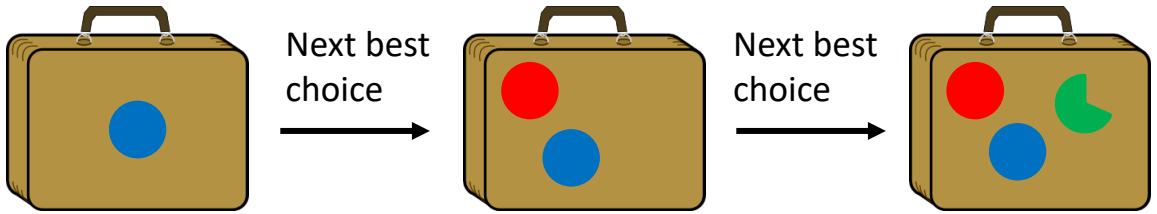
Greedy vs. Dynamic Programming

0/1 Knapsack
(using dynamic programming)



$T(n) = \Theta(nW)$

Fractional Knapsack
(using greedy approach)



$T(n) = \Theta(n \lg n)$

Matrix Chain Multiplication

- **Problem:** Given a sequence of matrices, output their product.
- **Input:** A sequence of n matrices $(M_1, \dots, M_n)$ such that for every pair (i, j) where $i < j$, if the dimension of M_i is (n_i, m_i) and the dimension of M_j is (n_j, m_j) then $m_i = n_j$
- **Output:** A sequence of matrix multiplications that minimizes the number of multiplication operations.

Matrix Chain Multiplication

- To multiply an $n \times m$ matrix A with a $m \times p$ matrix B to get a $n \times p$ matrix C we use the formula:

$$C_{i,j} = \sum_{k=1}^{m} A_{i,k} B_{k,j}$$

- Our complexity measure is the number of multiplications:
  - m multiplications are needed to compute each $C_{i,j}$
  - $(n \times m) \times p$ multiplications are needed to compute all elements of C

Matrix Chain Multiplication

- **Example**: Suppose we are trying to compute the product of the 3-matrix sequence (X, Y, Z)
  - Dimensions of X is 100×1
  - Dimensions of Y is 1×100
  - Dimensions of Z is 100×1
- One Solution:
  - Multiply X and Y to get a 100×100 matrix XY
    - No. of multiplications $= 100 \times 1 \times 100 = 10{,}000$
  - Multiply XY and Z to get the 100×1 result
    - No. of multiplications $= 100 \times 100 \times 1 = 10{,}000$
  - Total cost: $20{,}000$ multplications

Matrix Chain Multiplication

- **Example**: Suppose we are trying to compute the product of the 3-matrix sequence (X, Y, Z)
  - Dimensions of X is 100×1
  - Dimensions of Y is 1×100
  - Dimensions of Z is 100×1
- Another Solution:
  - Multiply Y and Z to get a 1×1 matrix YZ
    - No. of multiplications $= 1 \times 100 \times 1 = 100$
  - Multiply X and YZ to get the 100×1 result
    - No. of multiplications $= 100 \times 1 \times 1 = 100$
  - Total cost: 200 multplications

Matrix Chain Multiplication

- **Example**: Suppose we are trying to compute the product of the 4-matrix sequence (X, Y, Z, W)
  - Dimensions of X is 10×20
  - Dimensions of Y is 20×50
  - Dimensions of Z is 50×1
  - Dimensions of W is 1×100
- To represent the order of the multiplication, we use parentheses:
  - $X \times (Y \times (Z \times W))$ leads to 125,000 multiplications
  - $\big(X \times (Y \times Z)\big) \times W$ leads to 2,200 multiplications

Matrix Chain Multiplication

- **Naïve Approach:** Try all possible ways of parenthesization.
  - Is this a good idea?

- Let $P(n)$ be the number of different ways to parenthesize n matrices. Then:

$$P(n) = \begin{cases} \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \\ 1 & \text{if } n = 1 \end{cases}$$

- This recurrence is related to the Catalan numbers and solves to

$$P(n) = O\left(4^n / n^{1.5}\right)$$

Exponential!

Matrix Chain Multiplication

- Let us try to apply DP to solve this using polynomial time.
- Let $M_1, \ldots, M_n$ be the n matrices whose product $M_1 \times \cdots \times M_n$ we want to compute.
- Denote the dimensions of matrix M_i by $p_{i-1} \times p_i$
- Let $M_{i,j}$ where $i \leq j$ be the matrix corresponding to the partial result $M_i \times M_{i+1} \times \cdots \times M_j$

Matrix Chain Multiplication

- **Steps for Applying Dynamic Programming:**

1. Identify optimal substructure with overlapping subproblems.
2. Define a recursive formulation
   - Assuming solutions to sub-problems, recursively define the final solution
3. Compute the value of the optimal solution bottom-up
   - Store the solutions of sub-problems in a table (memorization) and build the final solution out of the stored values.

Matrix Chain Multiplication

- Any **optimal** parenthesization of $M_i \times M_{i+1} \times \cdots \times M_j$ must split the product into two parts:

$$M_i \times M_{i+1} \times \cdots \times M_k \text{ and } M_{k+1} \times M_{k+2} \times \cdots \times M_j$$

  for some $k \in [i, j]$
- What is the cost of such split:
  - Cost = cost of computing $M_{i,k}$ + cost of computing $M_{k+1,j}$ + cost of computing $M_{i,k}M_{k+1,j}$

1. Do we have optimal substructure?
2. Do we have overlapping subproblems?

Matrix Chain Multiplication

1. Do we have optimal substructure?
   - Yes!

- Proof sketch: suppose not. If the parenthesization of each of the subproducts is not optimal then we could substitute a better one and obtain a solution to the original problem with smaller cost.
- Thus the problem has optimal substructure.

Matrix Chain Multiplication

2. Do we have overlapping subproblems?
   - Yes!

- Proof sketch: Recall that there are $P(n) = O(4^n/n^{1.5})$ different ways to parenthesize (i.e. that many different subproblems), but there are $\Theta(n^2)$ **unique** subproblems since a subproblem of the form $M_{i,j}$ and $i, j \in [1, n]$.
- Thus we are solving some subproblems multiple times.
- By tabulating the solutions, we can solve every subproblem just once.

Matrix Chain Multiplication

- Having defined the type of subproblems it is very easy to recursively define the cost of the optimal solution.
  - Let $m_{i,j}$ be the minimum number of multiplications for computing the product $M_{i,j} = M_i \times M_{i+1} \times \cdots \times M_j$
  - Our goal is to compute $m_{1,n}$

- How do we define $m_{i,j}$ recursively?
  - Hint: consider the cases $i = j$ and $i \neq j$

Matrix Chain Multiplication

- Case 1: $i = j$
  - The product $M_{i,j} = M_i \times M_{i+1} \times \cdots \times M_j$ consists of just one matrix, so there is nothing to multiply.
  - The cost $m_{i,j} = 0$
- Case 2: $i \neq j$
  - We use the optimal substructure property
  - There is going to be some index k such that the product $M_i \times M_{i+1} \times \cdots \times M_j$ is split into two parts $M_i \times M_{i+1} \times \cdots \times M_k$ and $M_{k+1} \times M_{k+2} \times \cdots \times M_j$
  - The cost of this split will be $m_{i,j} = m_{i,k} + m_{k+1,j} + p_{i-1}p_kp_j$
  - How do we find the best split?

Matrix Chain Multiplication

- Case 2: $i \neq j$
  - How do we find the best split?

- Try all possible k and take the minimum
- Thus, the recursive formulation is:

$$m_{i,j} = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \leq k < j} \{m_{i,k} + m_{k+1,j} + p_{i-1}p_k p_j\} & \text{if } i < j \end{cases}$$

Matrix Chain Multiplication

$$m_{i,j} = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m_{i,k} + m_{k+1,j} + p_{i-1}p_k p_j\} & \text{if } i < j \end{cases}$$

Matrix-Chain $-$ **Order**$(M_1, \dots M_n)$:
for $i = 1$ **to** n
    $m[i,i] = 0$
for $l = 1$ **to** $n - 1$
    for $i = 1$ **to** $n - l$
        $j = i + l$
        $m[i,j] = \min\limits_{k}\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\}$
return $m[1,n]$

Runtime: $O(n^3)$

Matrix Chain Multiplication

- **Example:**

$m[i,j]$

$i \backslash j$	1	2	3	4
1	0			
2		0		
3			0	
4				0

Let $N = 4$ and the dimensions are:
$[10 \times 20], [20 \times 50], [50 \times 1]$ and $[1 \times 100]$

$$m_{i,j} = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m_{i,k} + m_{k+1,j} + p_{i-1}p_kp_j\} & \text{if } i < j \end{cases}$$

Matrix Chain Multiplication

- **Example:**

Let $N = 4$ and the dimensions are:
$[10 \times 20], [20 \times 50], [50 \times 1]$ and $[1 \times 100]$

$m[i,j]$

$i \backslash j$	1	2	3	4
1	0	10000	1200	
2		0	1000	
3			0	5000
4				0

$$m_{1,3} = \min \begin{cases} m_{1,1} + m_{2,3} + 10(20)(1) = 1200 \\ m_{1,2} + m_{3,3} + 10(50)(1) = 10500 \end{cases}$$

$$m_{i,j} = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m_{i,k} + m_{k+1,j} + p_{i-1}p_kp_j\} & \text{if } i < j \end{cases}$$

Matrix Chain Multiplication

- **Example:**

Let $N = 4$ and the dimensions are:
$[10 \times 20], [20 \times 50], [50 \times 1]$ and $[1 \times 100]$

$m[i,j]$

$i \backslash j$	1	2	3	4
1	0	10000	1200	
2		0	1000	3000
3			0	5000
4				0

$$m_{2,4} = \min \begin{cases} m_{2,2} + m_{3,4} + 20(50)(100) = 105000 \\ m_{2,3} + m_{4,4} + 20(1)(100) = 3000 \end{cases}$$

$$m_{i,j} = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m_{i,k} + m_{k+1,j} + p_{i-1}p_kp_j\} & \text{if } i < j \end{cases}$$

Matrix Chain Multiplication

- **Example:**

Let $N = 4$ and the dimensions are:
$[10 \times 20], [20 \times 50], [50 \times 1]$ and $[1 \times 100]$

$m[i,j]$

$i \backslash j$	1	2	3	4
1	0	10000	1200	2200
2		0	1000	3000
3			0	5000
4				0

$$m_{1,4} = \min \begin{cases} m_{1,1} + m_{2,4} + 10(20)(100) = 23000 \\ m_{1,2} + m_{3,4} + 10(50)(100) = 65000 \\ m_{1,3} + m_{4,4} + (10)(1)(100) = 2200 \end{cases}$$

$$m_{i,j} = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m_{i,k} + m_{k+1,j} + p_{i-1}p_k p_j\} & \text{if } i < j \end{cases}$$

Dynamic Programming

- Bottom-up vs. Top-down
  - If all subproblems must be solved at least once, **bottom-up** is better by a constant factor due to **no recursive** involvement.
  - If some subproblems may not need to be solved, **top-down** may be more efficient, since it only solves these subproblems which are definitely required.

Greedy vs. Dynamic Programming

- Dynamic Programming
  - Exhaustive – solves every possible subproblem exactly once.
  - Always arrives at an optimum solution
  - Often slower
  - Usually bottom-up

Property 1: Optimal Substructure
Property 2: Overlapping Subproblems

- Greedy
  - Makes locally optimal choices without looking at solutions to previous subproblems
  - Does not always give an optimum solution
  - Often more efficient
  - Usually top-down

Property 1: Optimal Substructure
Property 2: Greedy Choice