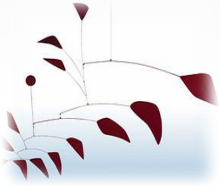


EECS 3101 - Design and Analysis of Algorithms

Shahin Kamali

Topic 4 - Dynamic Programming

Overview

- Dynamic Programming Framework & Applications
- Rod Cutting
- Matrix Chain Multiplication
- Longest Common Subsequence

Dynamic Programming Overview

- **Dynamic Programming** is similar to Divide & Conquer in the sense that it solves a problem by combining the solutions for subproblems.
  - Divide & Conquer solves subproblems **independently**.
  - Dynamic Programming applies when subproblems overlap, i.e., they share subsubproblems!

Dynamic Programming Overview

- **Dynamic Programming** is similar to Divide & Conquer in the sense that it solves a problem by combining the solutions for subproblems.
  - Divide & Conquer solves subproblems **independently**.
  - Dynamic Programming applies when subproblems overlap, i.e., they share subsubproblems!
- Dynamic Programming solves each subsubproblem just once and then saves it in a **table**
  - We avoid work of recomputing answers for subsubproblems.
  - **Programming** in this context refers to a tabular method, not to writing computer code.

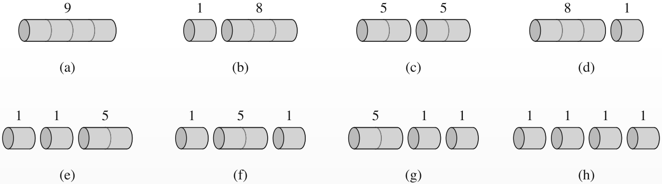
Dynamic Programming Overview

- Steps for designing a Dynamic Programming algorithm:
    1. Characterize the structure of an optimal solution.
    2. Recursively define the value of an optimal solution.
    3. Compute the value of an optimal solution, typically in a bottom-up fashion, and store results in a table.
    4. Construct an optimal solution from computed information in the table.

Rod Cutting

- You have a rod of length n, and you want to cut up the rod and sell the pieces ina way that maximizes the total amount of money you get. A piece of length i is worth p_i dollars.
    - E.g., for $n = 4$ and the following length/value table, we have 8 possible ways of cutting the rod, and the optimal cutting has value 10.

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



Inspecting the Problem

- How many ways are there to cut up a rod of length n?

Inspecting the Problem

- How many ways are there to cut up a rod of length n?
    - Roughly 2^{n-1}, because there are $n - 1$ places where we can choose to make cuts, and at each place, we either make a cut or we do not make a cut.

Inspecting the Problem

- How many ways are there to cut up a rod of length n?
  - Roughly 2^{n-1}, because there are $n-1$ places where we can choose to make cuts, and at each place, we either make a cut or we do not make a cut.
  - An exhaustive algorithm which tries all partitions runs in exponential time.

Basic Approach

- Rod cutting is a typical **optimization problem**, where we want to find to maximize a profit (or minimize a cost).

Basic Approach

- Rod cutting is a typical **optimization problem**, where we want to find to maximize a profit (or minimize a cost).
- For optimization problems, first, we ask “what is the maximum amount of profit we can get? (or minimum cost)"
    - Later we will extend the algorithm to give us the actual rod decomposition that leads to that maximum value.

Basic Approach

- Rod cutting is a typical **optimization problem**, where we want to find to maximize a profit (or minimize a cost).
- For optimization problems, first, we ask “what is the maximum amount of profit we can get? (or minimum cost)"
    - Later we will extend the algorithm to give us the actual rod decomposition that leads to that maximum value.
- **This general approach applies to almost all Dynamic Programming algorithms.**

Recursive Formulation

- Let r_i be the maximum amount of money you can get with a rod of size i. We can view the problem recursively as follows:
    - First, cut a piece off the left end of the rod, and sell it.
    - Then, find the optimal way to cut the remainder of the rod.

Recursive Formulation

- Let r_i be the maximum amount of money you can get with a rod of size i. We can view the problem recursively as follows:
    - First, cut a piece off the left end of the rod, and sell it.
    - Then, find the optimal way to cut the remainder of the rod.
- Now we don't know how large a piece we should cut off $\rightarrow$ try all possible cases.
    - First, try cutting a piece of length 1, and combining it with the optimal way to cut a rod of length $n-1$.

Recursive Formulation

- Let r_i be the maximum amount of money you can get with a rod of size i. We can view the problem recursively as follows:
  - First, cut a piece off the left end of the rod, and sell it.
  - Then, find the optimal way to cut the remainder of the rod.
- Now we don't know how large a piece we should cut off → try all possible cases.
  - First, try cutting a piece of length 1, and combining it with the optimal way to cut a rod of length $n - 1$.
  - Then try cutting a piece of length 2, and combining it with the optimal

Recursive Formulation

- Let r_i be the maximum amount of money you can get with a rod of size i. We can view the problem recursively as follows:
  - First, cut a piece off the left end of the rod, and sell it.
  - Then, find the optimal way to cut the remainder of the rod.
- Now we don't know how large a piece we should cut off → try all possible cases.
  - First, try cutting a piece of length 1, and combining it with the optimal way to cut a rod of length $n - 1$.
  - Then try cutting a piece of length 2, and combining it with the optimal way to cut a rod of length $n - 2$, and so on.
  - We try all the possible lengths and then pick the best one. We end up with the following: (when $i = n$, the rod is not cut at all)

$$r_n = \max_{1 \le i \le n} \{p_i + r_{n-i}\} \quad r_0 = 0$$

Recursive Formulation

- Let r_i be the maximum amount of money you can get with a rod of size i. We can view the problem recursively as follows:
  - First, cut a piece off the left end of the rod, and sell it.
  - Then, find the optimal way to cut the remainder of the rod.
- Now we don't know how large a piece we should cut off → try all possible cases.
  - First, try cutting a piece of length 1, and combining it with the optimal way to cut a rod of length $n-1$.
  - Then try cutting a piece of length 2, and combining it with the optimal way to cut a rod of length $n-2$, and so on.
  - We try all the possible lengths and then pick the best one. We end up with the following: (when $i = n$, the rod is not cut at all)

$$r_n = \max_{1 \le i \le n} \{p_i + r_{n-i}\} \qquad r_0 = 0$$

Recursive Implementation

- How should we compute $r_n = \max_{1 \le i \le n} \{p_i + r_{n-i}\} \quad r_0 = 0$?
  - The formula immediately translates into a recursive algorithm.

```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n − i))
return q
```

Recursive Implementation

- How should we compute $r_n = \max_{1 \le i \le n} \{p_i + r_{n-i}\} \quad r_0 = 0$?
  - The formula immediately translates into a recursive algorithm.

```
CUT-ROD($p, n$)
if $n == 0$
    return 0
$q = -\infty$
for $i = 1$ to $n$
    $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$
return $q$
```

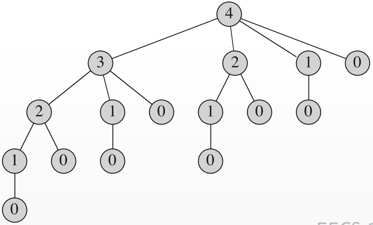
  - Is this good?

Recursive Implementation

- How should we compute $r_n = \max_{1 \le i \le n} \{p_i + r_{n-i}\} \quad r_0 = 0$?

```
CUT-ROD(p, n)
if n == 0
    return 0
$q = -\infty$
for i = 1 to n
    $q = \max(q, p[i] + \text{CUT-ROD}(p, n-i))$
return q
```

- There are many **repeated computation** in the recursion tree!

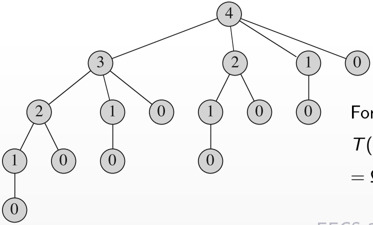


Recursive Implementation

- How should we compute $r_n = \max_{1 \le i \le n} \{p_i + r_{n-i}\} \quad r_0 = 0$?

```
CUT-ROD($p, n$)
if $n == 0$
    return 0
$q = -\infty$
for $i = 1$ to $n$
    $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$
return $q$
```

- There are many **repeated computation** in the recursion tree!



For the running time, we can write:

$T(n) > 2T(n-2) > 4T(n-4) > \ldots > 2^{n/2}T(1)$

$= \Omega(2^{n/2})$.

DP: Memoization (Top Down)

- How should we compute $r_n = \max_{1 \le i \le n} \{p_i + r_{n-i}\} \quad r_0 = 0$?
- We can store the result of the recursive calls, and if we need the result in a future recursive call, we can use the precomputed value. The answer will be stored in $r[n]$.

```
MEMOIZED-CUT-ROD($p, n$)
let $r[0..n]$ be a new array
for $i = 0$ to $n$
    $r[i] = -\infty$
return MEMOIZED-CUT-ROD-AUX($p, n, r$)
```

```
MEMOIZED-CUT-ROD-AUX($p, n, r$)
if $r[n] \geq 0$
    return $r[n]$
if $n == 0$
    $q = 0$
else $q = -\infty$
    for $i = 1$ to $n$
        $q = \max(q, p[i] + $MEMOIZED-CUT-ROD-AUX$(p, n - i, r))$
$r[n] = q$
return $q$
```

DP: Memoization (Top Down)

- How should we compute $r_n = \max\limits_{1 \leq i \leq n} \{p_i + r_{n-i}\} \quad r_0 = 0$?
- We can store the result of the recursive calls, and if we need the result in a future recursive call, we can use the precomputed value. The answer will be stored in $r[n]$.
    - Each subproblem is solved exactly once. For a subproblem of size i, we spend $\Theta(i)$ (we run through i iterations of the for loop) → The running time is $\Theta(n) + \Theta(n-1) + \ldots + \Theta(1) = \Theta(n^2)$.

```
MEMOIZED-CUT-ROD(p, n)
let r[0..n] be a new array
for i = 0 to n
    r[i] = -∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0
    return r[n]
if n == 0
    q = 0
else q = -∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r))
r[n] = q
return q
```

DP: Memoization (Bottom Up)

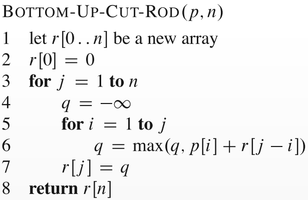
- How should we compute $r_n = \max_{1 \leq i \leq n} \{p_i + r_{n-i}\} \quad r_0 = 0$?
- We proactively compute the solutions for smaller rods first, knowing that they will later be used to compute the solutions for larger rods. The answer will be stored in $r[n]$.
  - **Most people will write the bottom up procedure when they implement a dynamic programming algorithm.**

```
BOTTOM-UP-CUT-ROD($p, n$)
let $r[0..n]$ be a new array
$r[0] = 0$
for $j = 1$ to $n$
    $q = -\infty$
    for $i = 1$ to $j$
        $q = \max(q, p[i] + r[j-i])$
    $r[j] = q$
return $r[n]$
```

DP: Memoization (Bottom Up)

- How should we compute $r_n = \max_{1 \leq i \leq n} \{p_i + r_{n-i}\} \quad r_0 = 0$?
- We proactively compute the solutions for smaller rods first, knowing that they will later be used to compute the solutions for larger rods. The answer will be stored in $r[n]$.
  - The running time is still $\Theta(n^2)$.

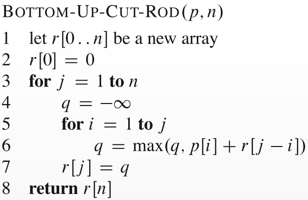
  - **Most people will write the bottom up procedure when they implement a dynamic programming algorithm.**



```
BOTTOM-UP-CUT-ROD$(p, n)$
let $r[0\,..\,n]$ be a new array
$r[0] = 0$
for $j = 1$ to $n$
    $q = -\infty$
    for $i = 1$ to $j$
        $q = \max(q, p[i] + r[j - i])$
    $r[j] = q$
return $r[n]$
```

DP: Memoization (Bottom Up)

- How should we compute $r_n = \max_{1 \le i \le n} \{p_i + r_{n-i}\} \quad r_0 = 0$?
- We proactively compute the solutions for smaller rods first, knowing that they will later be used to compute the solutions for larger rods. The answer will be stored in $r[n]$.
    - The running time is still $\Theta(n^2)$.
    - Often the bottom up approach is simpler to write, and has less overhead, because you don't have to keep a recursive call stack.
    - **Most people will write the bottom up procedure when they implement a dynamic programming algorithm.**



```
BOTTOM-UP-CUT-ROD$(p, n)$
let $r[0..n]$ be a new array
$r[0] = 0$
for $j = 1$ to $n$
    $q = -\infty$
    for $i = 1$ to $j$
        $q = \max(q, p[i] + r[j-i])$
    $r[j] = q$
return $r[n]$
```

Reconstructing a solution

- If we want to actually find **the optimal way to split the rod**, instead of just the maximum profit we can get, we can create another array s:
  - $s[j] = i$ iff the best thing to do when we have a rod of length j is to cut off a piece of length i.
  - Using these values $s[j]$, we can reconstruct the optimal rod decomposition.

```
EXTENDED-BOTTOM-UP-CUT-ROD($p, n$)
let $r[0..n]$ and $s[0..n]$ be new arrays
$r[0] = 0$
for $j = 1$ to $n$
    $q = -\infty$
    for $i = 1$ to $j$
        if $q < p[i] + r[j-i]$
            $q = p[i] + r[j-i]$
            $s[j] = i$
    $r[j] = q$
return $r$ and $s$
```

```
PRINT-CUT-ROD-SOLUTION($p, n$)
$(r, s) =$ EXTENDED-BOTTOM-UP-CUT-ROD($p, n$)
while $n > 0$
    print $s[n]$
    $n = n - s[n]$
```

The Example Problem's Answer

- For our example, the program produces this answer:

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

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

Dynamic programming remarks

- **Optimal substructure:** To solve a optimization problem using dynamic programming, we must first **characterize the structure of an optimal solution.**
    - Specifically, we must prove that we can **create an optimal solution to a problem using optimal solutions to smaller subproblems.**
    - Then, we can store optimal solutions for all subproblems in a table → compute later elements in the table from earlier elements in the table.

Dynamic programming remarks

- **Optimal substructure:** To solve a optimization problem using dynamic programming, we must first **characterize the structure of an optimal solution.**
    - Specifically, we must prove that we can **create an optimal solution to a problem using optimal solutions to smaller subproblems.**
    - Then, we can store optimal solutions for all subproblems in a table → compute later elements in the table from earlier elements in the table.
    - If the optimal solution to a problem might not require subproblem solutions to be optimal, then we cannot use dynamic programming.

Dynamic programming remarks

- **Overlapping Subproblems**
  - For dynamic programming to be useful, the recursive algorithm should require us to compute optimal solutions to the same subproblems over and over again → Then we benefit from just computing them once and then using the results later.
  - In total, there should be **a small number of distinct subproblems** (i.e. polynomial in the input size), even if there is an exponential number of total subproblems.

Longest common subsequence

- We are given two sequences X and Y , and want to find the longest possible sequence that is a subsequence of both X and Y.
- E.g., for $X = ABCBDAB$ and $Y = BDCABA$:
    - BCA is a common sequence of both X and Y.

Longest common subsequence

- We are given two sequences X and Y , and want to find the longest possible sequence that is a subsequence of both X and Y.
- E.g., for $X = ABCBDAB$ and $Y = BDCABA$:
  - BCA is a common sequence of both X and Y.
  - $BCBA$ is a longer sequence that is also common to both X and Y.

Longest common subsequence

- We are given two sequences X and Y , and want to find the longest possible sequence that is a subsequence of both X and Y.
- E.g., for $X = ABCBDAB$ and $Y = BDCABA$:
  - BCA is a common sequence of both X and Y.
  - $BCBA$ is a longer sequence that is also common to both X and Y.
  - Both $BCBA$ and $BDAB$ are longest common subsequences, since there are no common sequences of length 5 or greater

LCS Algorithms

- if $|X| = m$, $|Y| = n$, then there are 2^m subsequences of X; we must compare each with Y (n comparisons)
  - So the running time of the brute-force algorithm is $O(n2^m)$.
- Notice that the LCS problem has optimal substructure: solutions of subproblems are parts of the final solution → should we use dynamic programming?

Optimal substructures

- The first step use dynamic programming is create an optimal solution to this problem using optimal solutions to subproblems → **a recursive formulation of the optimal solution**.
- The hardest part is to decide what the subproblems are. For the LCS we have two cases:
  - **Case 1:** The last elements of X and Y are equal.
  - **Case 2:** The last elements of X and Y are not equal.

LCS Optimal Formulation

- **Case 1:** The last elements of X and Y are equal.

$$X = \text{ABCBDA}\textbf{B} \text{ and } \ Y = \text{BDCA}\textbf{B}$$

.

LCS Optimal Formulation

- **Case 1:** The last elements of X and Y are equal.

$$X = \text{ABCBDA}\mathbf{B} \text{ and } \; Y = \text{BDCA}\mathbf{B}$$

.

- Then the last element must both be part of the longest common subsequence
- We can chop both elements off the ends of the subsequence (adding them to a common subsequence) and find the longest common subsequence of the smaller sequences.
- The LCS of $X = ABCBDA\mathbf{B}$ and $Y = BDCA\mathbf{B}$ can be formed by finding the LCS of $ABCBDA$ and $BDCA$, which is BDA, and adding B to it, that is LCS of X and Y is $BDAB$.

LCS Optimal Formulation

- **Case 2:** The last elements of X and Y are not equal.

$$X = \text{ABCBDAB}\textbf{A} \text{ and } \; Y = \text{BDCA}\textbf{B}$$

.

LCS Optimal Formulation

- **Case 2:** The last elements of X and Y are not equal.

$$X = \text{ABCBDAB}\mathbf{A} \text{ and } \quad Y = \text{BDCA}\mathbf{B}$$

.

- Either the last element of X or the last element of Y cannot be part of the longest common subsequence.
- we can find the LCS of X and a smaller version of Y in which the last element is missing, or the LCS of Y and a smaller version of X in which the last element is missing.
- The LCS of $X = ABCBDAB\mathbf{A}$ and $Y = BDCA\mathbf{B}$ can be formed by:
    - The LCS of $ABCBDAB\mathbf{A}$ and $BDCA$, which is BCA.
    - The LCS of $ABCBDAB$ and $BDCA\mathbf{B}$, which is $BDAB$.
    - Taking the LCS with max length, i.e., $BDAB$.

LCS Dynamic Programming Solution

- First we'll find the length of LCS. Later we'll modify the algorithm to find LCS itself.
- Define X_i, Y_j to be the prefixes of X and Y of length i and j, respectively.
- Define $c[i, j]$ to be the length of LCS of X_i and Y_j Then the length of LCS of X and Y will be $c[m, n]$.
    - Since X_0 and Y_0 are empty strings, their LCS is always empty (i.e. $c[0, 0] = 0$)
    - LCS of empty string and any other string is empty, so for every i and j we have $c[0, j] = c[i, 0] = 0$.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j\,, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j\,. \end{cases}$$

LCS Dynamic Programming Solution

- **Optimal Substructure:** we have characterized the optimal solution recursively using optimal solutions to smaller problems.
- **Overlapping Subproblems:** How many subproblems exist?
  - Each $c[i, j]$ is associated with one sub-problem that asks for LCS of X_i and Y_j → There are $\Theta(m.n)$ subproblems.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j\,, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j\,. \end{cases}$$

LCS Dynamic Programming Solution

- Using the recurrence, we can write the actual pseudocode.
    - We populate the table in a certain order, because some elements depend on other elements of the table having already been computed

```
LCS-LENGTH($X, Y$)
$m = X.length$
$n = Y.length$
let $b[1..m, 1..n]$ and $c[0..m, 0..n]$ be new tables
for $i = 1$ to $m$
    $c[i, 0] = 0$
for $j = 0$ to $n$
    $c[0, j] = 0$
for $i = 1$ to $m$
    for $j = 1$ to $n$
        if $x_i == y_j$
            $c[i, j] = c[i-1, j-1] + 1$
            $b[i, j] =$ "↖"
        elseif $c[i-1, j] \geq c[i, j-1]$
            $c[i, j] = c[i-1, j]$
            $b[i, j] =$ "↑"
        else $c[i, j] = c[i, j-1]$
            $b[i, j] =$ "←"
return $c$ and $b$
```

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
    - We first fill the first column and row (at index 0).

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i							
0	A							
1	B							
2	C							
3	B							
4	D							
5	A							
6	B							

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0						
1	B	0						
2	C	0						
3	B	0						
4	D	0						
5	A	0						
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0					
1	B	0						
2	C	0						
3	B	0						
4	D	0						
5	A	0						
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0				
1	B	0						
2	C	0						
3	B	0						
4	D	0						
5	A	0						
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0			
1	B	0						
2	C	0						
3	B	0						
4	D	0						
5	A	0						
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

j		0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1		
1	B	0						
2	C	0						
3	B	0						
4	D	0						
5	A	0						
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	
1	B	0						
2	C	0						
3	B	0						
4	D	0						
5	A	0						
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0						
2	C	0						
3	B	0						
4	D	0						
5	A	0						
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1					
2	C	0						
3	B	0						
4	D	0						
5	A	0						
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
    - We first fill the first column and row (at index 0).
    - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1				
2	C	0						
3	B	0						
4	D	0						
5	A	0						
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

j		0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑0	↑0	↑0	↖1	←1	↖1
1	B	0	↖1	←1	←1			
2	C	0						
3	B	0						
4	D	0						
5	A	0						
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

j		0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1		
2	C	0						
3	B	0						
4	D	0						
5	A	0						
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	
2	C	0						
3	B	0						
4	D	0						
5	A	0						
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0						
3	B	0						
4	D	0						
5	A	0						
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
    - We first fill the first column and row (at index 0).
    - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1					
3	B	0						
4	D	0						
5	A	0						
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
    - We first fill the first column and row (at index 0).
    - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1				
3	B	0						
4	D	0						
5	A	0						
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2			
3	B	0						
4	D	0						
5	A	0						
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

j		0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2		
3	B	0						
4	D	0						
5	A	0						
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	
3	B	0						
4	D	0						
5	A	0						
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
3	B	0						
4	D	0						
5	A	0						
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
3	B	0	↖ 1					
4	D	0						
5	A	0						
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑0	↑0	↑0	↖1	←1	↖1
1	B	0	↖1	←1	←1	↑1	↖2	←2
2	C	0	↑1	↑1	↖2	←2	↑2	↑2
3	B	0	↖1	↑1				
4	D	0						
5	A	0						
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
    - We first fill the first column and row (at index 0).
    - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
3	B	0	↖ 1	↑ 1	↑ 2			
4	D	0						
5	A	0						
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
3	B	0	↖ 1	↑ 1	↑ 2	↑ 2		
4	D	0						
5	A	0						
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
    - We first fill the first column and row (at index 0).
    - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
3	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	
4	D	0						
5	A	0						
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
3	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
4	D	0						
5	A	0						
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
3	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
4	D	0	↑ 1					
5	A	0						
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
3	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
4	D	0	↑ 1	↖ 2				
5	A	0						
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
3	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
4	D	0	↑ 1	↖ 2	↑ 2			
5	A	0						
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
3	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
4	D	0	↑ 1	↖ 2	↑ 2	↑ 2		
5	A	0						
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
    - We first fill the first column and row (at index 0).
    - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
3	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
4	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	
5	A	0						
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
3	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
4	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
5	A	0						
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
3	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
4	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
5	A	0	↑ 1					
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
    - We first fill the first column and row (at index 0).
    - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑0	↑0	↑0	↖1	←1	↖1
1	B	0	↖1	←1	←1	↑1	↖2	←2
2	C	0	↑1	↑1	↖2	←2	↑2	↑2
3	B	0	↖1	↑1	↑2	↑2	↖3	←3
4	D	0	↑1	↖2	↑2	↑2	↑3	↑3
5	A	0	↑1	↑2				
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
3	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
4	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
5	A	0	↑ 1	↑ 2	↑ 2			
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
3	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
4	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
5	A	0	↑ 1	↑ 2	↑ 2	↖ 3		
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

j		0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
3	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
4	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
5	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
3	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
4	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
5	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
6	B	0						

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
3	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
4	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
5	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
6	B	0	↖ 1					

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
3	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
4	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
5	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
6	B	0	↖ 1	↑ 2				

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑0	↑0	↑0	↖1	←1	↖1
1	B	0	↖1	←1	←1	↑1	↖2	←2
2	C	0	↑1	↑1	↖2	←2	↑2	↑2
3	B	0	↖1	↑1	↑2	↑2	↖3	←3
4	D	0	↑1	↖2	↑2	↑2	↑3	↑3
5	A	0	↑1	↑2	↑2	↖3	↑3	↖4
6	B	0	↖1	↑2	↑2			

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
    - We first fill the first column and row (at index 0).
    - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑0	↑0	↑0	↖1	←1	↖1
1	B	0	↖1	←1	←1	↑1	↖2	←2
2	C	0	↑1	↑1	↖2	←2	↑2	↑2
3	B	0	↖1	↑1	↑2	↑2	↖3	←3
4	D	0	↑1	↖2	↑2	↑2	↑3	↑3
5	A	0	↑1	↑2	↑2	↖3	↑3	↖4
6	B	0	↖1	↑2	↑2	↑3		

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
3	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
4	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
5	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
6	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - We first fill the first column and row (at index 0).
  - The remaining indices are filled row by row.

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
3	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
4	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
5	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
6	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
    - After filling the table, we use arrows to detect the LCS (formed by indices at which the arrow points to Top and Left)

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
3	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
4	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
5	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
6	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - After filling the table, we use arrows to detect the LCS (formed by indices at which the arrow points to Top and Left)

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
3	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
4	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
5	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
6	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - After filling the table, we use arrows to detect the LCS (formed by indices at which the arrow points to Top and Left)

	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
3	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
4	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
5	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
6	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - After filling the table, we use arrows to detect the LCS (formed by indices at which the arrow points to Top and Left)

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
3	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
4	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
5	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
6	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

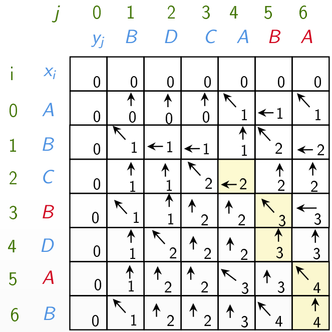
LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - After filling the table, we use arrows to detect the LCS (formed by indices at which the arrow points to Top and Left)

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
3	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
4	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
5	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
6	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - After filling the table, we use arrows to detect the LCS (formed by indices at which the arrow points to Top and Left)



	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
3	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
4	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
5	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
6	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

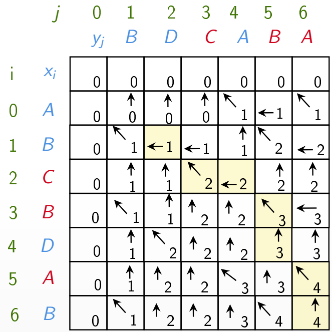
LCS Dynamic Programming Example

- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
    - After filling the table, we use arrows to detect the LCS (formed by indices at which the arrow points to Top and Left)

	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
3	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
4	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
5	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
6	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

LCS Dynamic Programming Example

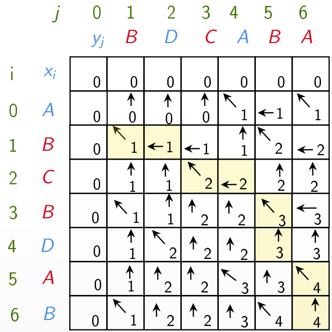
- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - After filling the table, we use arrows to detect the LCS (formed by indices at which the arrow points to Top and Left)



	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
3	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
4	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
5	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
6	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

LCS Dynamic Programming Example

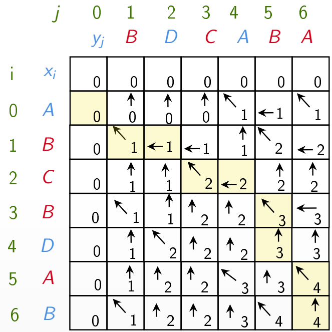
- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - After filling the table, we use arrows to detect the LCS (formed by indices at which the arrow points to Top and Left)



j		0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
3	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
4	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
5	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
6	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

LCS Dynamic Programming Example

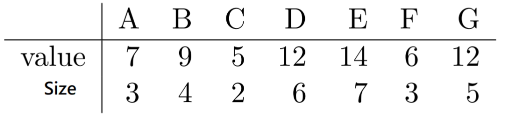
- Let's see how LCS algorithm works when $X = ABCBDAB$ and $Y = BDCABA$
  - After filling the table, we use arrows to detect the LCS (formed by indices at which the arrow points to Top and Left)



	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
i	x_i	0	0	0	0	0	0	0
0	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
1	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
2	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
3	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
4	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
5	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
6	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Knapsack Problem

- In the **0-1 knapsack problem**, we are given a set of n items $a_1, \ldots, a_n$.
  - Each item a_i has a size s_i and a value v_i.
  - We are also given a size bound S (the capacity of our knapsack).
  - The goal is to find the subset of items of maximum total value such that sum of their sizes is at most S (they all fit into the knapsack).
    - In the example below, where $S = 15$, the optimal strategy is to do parts A, B, F, and G for a total of 34 points.



	A	B	C	D	E	F	G
value	7	9	5	12	14	6	12
Size	3	4	2	6	7	3	5

Greedy Strategy

- **Option 1:** process items in order $1, 2, \dots, n$, and accepts an item as long as it fits (first A, then B, etc.)
  - This selects A, B, C and D for a profit of 33 (which is not optimal because $\{A, B, F, G\}$ has profit 34)

	A	B	C	D	E	F	G
value	7	9	5	12	14	6	12
Size	3	4	2	6	7	3	5

Greedy Strategy

- **Option 2:** Sort items by their value-to-size ratio, process items in the sorted order, and accepts an item as long as it fits (first A, then B, etc.)
  - This selects $C(2.5), G(2.4), A(2.33)$, and $B(2.25)$ for a profit of 33 (which is not optimal because $\{A, B, F, G\}$ has profit 34.

	A	B	C	D	E	F	G
value	7	9	5	12	14	6	12
Size	3	4	2	6	7	3	5
	7/3	9/4	5/2	12/6	14/7	6/3	12/5

Dynamic Programming for Knapsack

- **Step 1:** Describe the optimal solution using the optimal solution for the subproblems.

Dynamic Programming for Knapsack

- **Step 1:** Describe the optimal solution using the optimal solution for the subproblems.
- Subproblem: finding the optimal profit (value) when items are $a_1, a_2, \ldots, a_k$ (for $k \leq n$), and the space is B (for $B \leq S$).
- Should I accept or reject a_k?
  - If I accept a_k, the optimal profit will be v_k plus the profit of placing $a_1, \ldots, a_{k-1}$ in a space of $B - s_k$.
  - If I reject a_k, the optimal profit will be the profit of placing $a_1, \ldots, a_{k-1}$ in a space of B.
  - If I have the solution for the two sub-problems, I can take the max between the two!

Dynamic Programming for Knapsack

- **Step 2:** Describe the value of the optimal solution recursively
- Let $V(k, B)$ denote the value of the highest value solution that uses items from among the set $1, 2, \ldots, k$ and uses space at most B.
    - We want to find the value of $V(n, S)$
    - Here is the recursive value for $V(k, B)$.

$$V(k,B) = \begin{cases} 0 & \text{if } k = 0 \\ V(k-1,B) & \text{if } s_k > B \\ \max\{v_k + V(k-1, B - s_k), V(k-1,B)\} & \text{otherwise} \end{cases}$$

Dynamic Programming for Knapsack

- **Step 3:** Fill the Dynamic Programming table (in a bottom-up way) to find $V(n, S)$.

$$V(k,B) = \begin{cases} 0 & \text{if } k = 0 \\ V(k-1,B) & \text{if } s_k > B \\ \max\{v_k + V(k-1, B - s_k), V(k-1,B)\} & \text{otherwise} \end{cases}$$

  - We fill the table row by row; the value of each row depends on the previous rows; The first row is all 0, $V(0, B) = 0$.

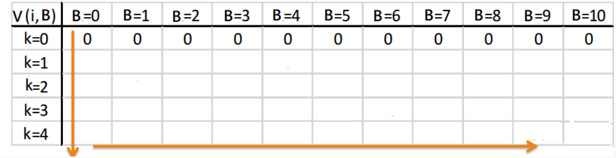
```
Knapsack ($s[], v[], n, S$)
    for $k = 0$ to $n$
        for $B = 0$ to $S$
            if $i = 0$
                $V(k, B) \leftarrow 0$
            else
                if $s_k > B$
                    $V(k, B) \leftarrow V(k-1, B)$
                else
                    $V(k, B) \leftarrow \max\{v_k + V(k-1, B - s_k)V(k-1, B)\}$
    return $V$
```

Dynamic Programming for Knapsack

- **Step 3:** Fill the Dynamic Programming table (in a bottom-up way) to find $V(n, S)$.

$$V(k,B) = \begin{cases} 0 & \text{if } k = 0 \\ V(k-1,B) & \text{if } s_k > B \\ \max\{v_k + V(k-1, B-s_k), V(k-1,B)\} & \text{otherwise} \end{cases}$$

  - We fill the table row by row; the value of each row depends on the previous rows; The first row is all 0, $V(0, B) = 0$.
    - Here, $S = 10$; item sizes are $(5, 4, 6, 3)$ and values are $(10, 40, 30, 50)$, that is, the first item has size 10 and value 5.



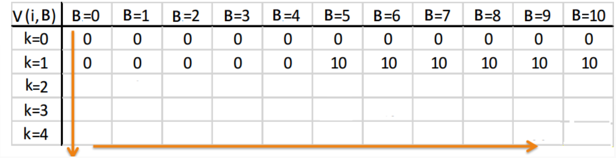
V(i,B)	B=0	B=1	B=2	B=3	B=4	B=5	B=6	B=7	B=8	B=9	B=10
k=0	0	0	0	0	0	0	0	0	0	0	0
k=1											
k=2											
k=3											
k=4											

Dynamic Programming for Knapsack

- **Step 3:** Fill the Dynamic Programming table (in a bottom-up way) to find $V(n, S)$.

$$V(k,B) = \begin{cases} 0 & \text{if } k = 0 \\ V(k-1,B) & \text{if } s_k > B \\ \max\{v_k + V(k-1, B-s_k), V(k-1,B)\} & \text{otherwise} \end{cases}$$

  - We fill the table row by row; the value of each row depends on the previous rows; The first row is all 0, $V(0, B) = 0$.
    - Here, $S = 10$; item sizes are $(5, 4, 6, 3)$ and values are $(10, 40, 30, 50)$, that is, the first item has size 10 and value 5.



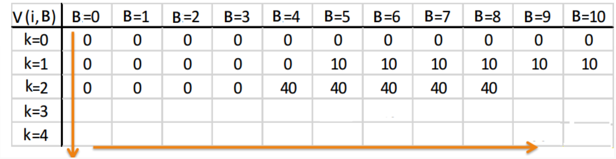
V(i,B)	B=0	B=1	B=2	B=3	B=4	B=5	B=6	B=7	B=8	B=9	B=10
k=0	0	0	0	0	0	0	0	0	0	0	0
k=1	0	0	0	0	0	10	10	10	10	10	10
k=2											
k=3											
k=4											

Dynamic Programming for Knapsack

- **Step 3:** Fill the Dynamic Programming table (in a bottom-up way) to find $V(n, S)$.

$$V(k,B) = \begin{cases} 0 & \text{if } k = 0 \\ V(k-1,B) & \text{if } s_k > B \\ \max\{v_k + V(k-1,B-s_k), V(k-1,B)\} & \text{otherwise} \end{cases}$$

  - We fill the table row by row; the value of each row depends on the previous rows; The first row is all 0, $V(0, B) = 0$.
    - Here, $S = 10$; item sizes are $(5, 4, 6, 3)$ and values are $(10, 40, 30, 50)$, that is, the first item has size 10 and value 5.



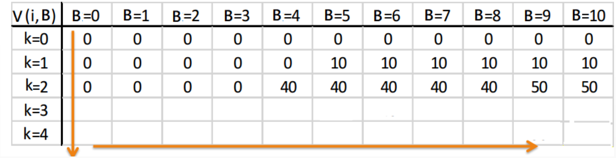
V(i,B)	B=0	B=1	B=2	B=3	B=4	B=5	B=6	B=7	B=8	B=9	B=10
k=0	0	0	0	0	0	0	0	0	0	0	0
k=1	0	0	0	0	0	10	10	10	10	10	10
k=2	0	0	0	0	40	40	40	40	40		
k=3											
k=4											

Dynamic Programming for Knapsack

- **Step 3:** Fill the Dynamic Programming table (in a bottom-up way) to find $V(n, S)$.

$$V(k, B) = \begin{cases} 0 & \text{if } k = 0 \\ V(k-1, B) & \text{if } s_k > B \\ \max\{v_k + V(k-1, B - s_k), V(k-1, B)\} & \text{otherwise} \end{cases}$$

  - We fill the table row by row; the value of each row depends on the previous rows; The first row is all 0, $V(0, B) = 0$.
    - Here, $S = 10$; item sizes are $(5, 4, 6, 3)$ and values are $(10, 40, 30, 50)$, that is, the first item has size 10 and value 5.
    - $V(2, 9) = \max\{v_2 + V(1, 9 - s_2) = 40 + 10, V(1, 9) = 10\} = 50$



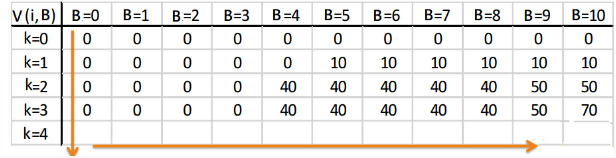
V(i,B)	B=0	B=1	B=2	B=3	B=4	B=5	B=6	B=7	B=8	B=9	B=10
k=0	0	0	0	0	0	0	0	0	0	0	0
k=1	0	0	0	0	0	10	10	10	10	10	10
k=2	0	0	0	0	40	40	40	40	40	50	50
k=3											
k=4											

Dynamic Programming for Knapsack

- **Step 3:** Fill the Dynamic Programming table (in a bottom-up way) to find $V(n, S)$.

$$V(k,B) = \begin{cases} 0 & \text{if } k = 0 \\ V(k-1,B) & \text{if } s_k > B \\ \max\{v_k + V(k-1, B-s_k), V(k-1,B)\} & \text{otherwise} \end{cases}$$

  - We fill the table row by row; the value of each row depends on the previous rows; The first row is all 0, $V(0, B) = 0$.
    - Here, $S = 10$; item sizes are $(5, 4, 6, 3)$ and values are $(10, 40, 30, 50)$, that is, the first item has size 10 and value 5.
    - $V(2, 9) = \max\{v_2 + V(1, 9 - s_2) = 40 + 10, V(1, 9) = 10\} = 50$



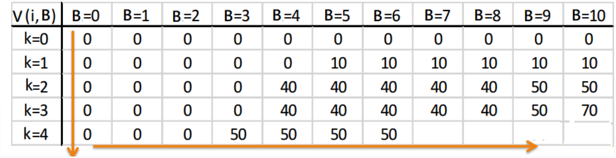
V(i,B)	B=0	B=1	B=2	B=3	B=4	B=5	B=6	B=7	B=8	B=9	B=10
k=0	0	0	0	0	0	0	0	0	0	0	0
k=1	0	0	0	0	0	10	10	10	10	10	10
k=2	0	0	0	0	40	40	40	40	40	50	50
k=3	0	0	0	0	40	40	40	40	40	50	70
k=4											

Dynamic Programming for Knapsack

- **Step 3:** Fill the Dynamic Programming table (in a bottom-up way) to find $V(n, S)$.

$$V(k,B) = \begin{cases} 0 & \text{if } k = 0 \\ V(k-1,B) & \text{if } s_k > B \\ \max\{v_k + V(k-1, B-s_k), V(k-1,B)\} & \text{otherwise} \end{cases}$$

  - We fill the table row by row; the value of each row depends on the previous rows; The first row is all 0, $V(0, B) = 0$.
    - Here, $S = 10$; item sizes are $(5, 4, 6, 3)$ and values are $(10, 40, 30, 50)$, that is, the first item has size 10 and value 5.
    - $V(2, 9) = \max\{v_2 + V(1, 9 - s_2) = 40 + 10, V(1, 9) = 10\} = 50$



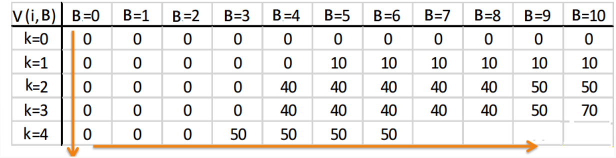
V(i,B)	B=0	B=1	B=2	B=3	B=4	B=5	B=6	B=7	B=8	B=9	B=10
k=0	0	0	0	0	0	0	0	0	0	0	0
k=1	0	0	0	0	0	10	10	10	10	10	10
k=2	0	0	0	0	40	40	40	40	40	50	50
k=3	0	0	0	0	40	40	40	40	40	50	70
k=4	0	0	0	50	50	50	50				

Dynamic Programming for Knapsack

- **Step 3:** Fill the Dynamic Programming table (in a bottom-up way) to find $V(n, S)$.

$$V(k,B) = \begin{cases} 0 & \text{if } k = 0 \\ V(k-1,B) & \text{if } s_k > B \\ \max\{v_k + V(k-1, B-s_k), V(k-1,B)\} & \text{otherwise} \end{cases}$$

  - We fill the table row by row; the value of each row depends on the previous rows; The first row is all 0, $V(0, B) = 0$.
    - Here, $S = 10$; item sizes are $(5, 4, 6, 3)$ and values are $(10, 40, 30, 50)$, that is, the first item has size 10 and value 5.
    - $V(2, 9) = \max\{v_2 + V(1, 9 - s_2) = 40 + 10, V(1, 9) = 10\} = 50$
    - $V(4, 7) = \max\{v_4 + V(3, 7 - s_4) = 50 + 40, V(3, 7) = 40\} = 90$



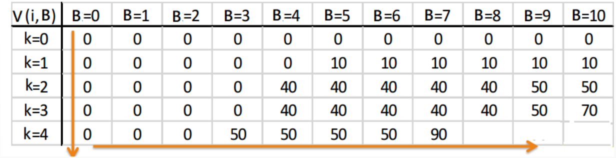
V(i,B)	B=0	B=1	B=2	B=3	B=4	B=5	B=6	B=7	B=8	B=9	B=10
k=0	0	0	0	0	0	0	0	0	0	0	0
k=1	0	0	0	0	0	10	10	10	10	10	10
k=2	0	0	0	0	40	40	40	40	40	50	50
k=3	0	0	0	0	40	40	40	40	40	50	70
k=4	0	0	0	50	50	50	50				

Dynamic Programming for Knapsack

- **Step 3:** Fill the Dynamic Programming table (in a bottom-up way) to find $V(n, S)$.

$$V(k,B) = \begin{cases} 0 & \text{if } k = 0 \\ V(k-1,B) & \text{if } s_k > B \\ \max\{v_k + V(k-1, B-s_k), V(k-1,B)\} & \text{otherwise} \end{cases}$$

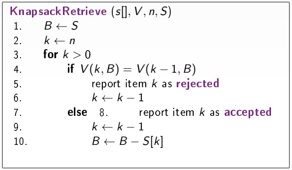
  - We fill the table row by row; the value of each row depends on the previous rows; The first row is all 0, $V(0, B) = 0$.
    - Here, $S = 10$; item sizes are $(5, 4, 6, 3)$ and values are $(10, 40, 30, 50)$, that is, the first item has size 10 and value 5.
    - $V(2, 9) = \max\{v_2 + V(1, 9 - s_2) = 40 + 10, V(1, 9) = 10\} = 50$
    - $V(4, 7) = \max\{v_4 + V(3, 7 - s_4) = 50 + 40, V(3, 7) = 40\} = 90$



V(i,B)	B=0	B=1	B=2	B=3	B=4	B=5	B=6	B=7	B=8	B=9	B=10
k=0	0	0	0	0	0	0	0	0	0	0	0
k=1	0	0	0	0	0	10	10	10	10	10	10
k=2	0	0	0	0	40	40	40	40	40	50	50
k=3	0	0	0	0	40	40	40	40	40	50	70
k=4	0	0	0	50	50	50	50	90			

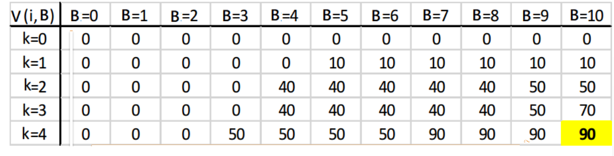
Dynamic Programming for Knapsack

- **Step 4:** Go backwards in the table to retrieve the accepted items.



```
KnapsackRetrieve ($s[], V, n, S$)
    $B \leftarrow S$
    $k \leftarrow n$
    for $k > 0$
        if $V(k, B) = V(k-1, B)$
            report item $k$ as rejected
            $k \leftarrow k - 1$
        else
            report item $k$ as accepted
            $k \leftarrow k - 1$
            $B \leftarrow B - S[k]$
```

- Here, $S = 10$; sizes are $(5, 4, 6, 3)$ and values are $(10, 40, 30, 50)$.
- First, $V[4, 10] = 90$ and $V[3, 10] = 70$; we can conclude a_4 is accepted. The remaining space would be $10 - s_4 = 7$. We should check $V[3, 7]$ and repeat; Accepted items are a_2 and a_4.



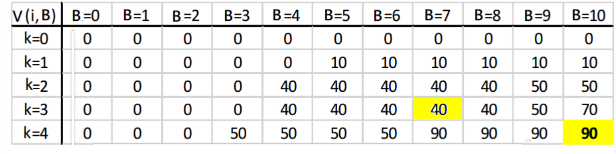
V(i,B)	B=0	B=1	B=2	B=3	B=4	B=5	B=6	B=7	B=8	B=9	B=10
k=0	0	0	0	0	0	0	0	0	0	0	0
k=1	0	0	0	0	0	10	10	10	10	10	10
k=2	0	0	0	0	40	40	40	40	40	50	50
k=3	0	0	0	0	40	40	40	40	40	50	70
k=4	0	0	0	50	50	50	50	90	90	90	**90**

Dynamic Programming for Knapsack

- **Step 4:** Go backwards in the table to retrieve the accepted items.

```
KnapsackRetrieve ($s[], V, n, S$)
    $B \leftarrow S$
    $k \leftarrow n$
    for $k > 0$
        if $V(k, B) = V(k-1, B)$
            report item $k$ as rejected
            $k \leftarrow k - 1$
        else
            report item $k$ as accepted
            $k \leftarrow k - 1$
            $B \leftarrow B - S[k]$
```

- Here, $S = 10$; sizes are $(5, 4, 6, 3)$ and values are $(10, 40, 30, 50)$.
- First, $V[4, 10] = 90$ and $V[3, 10] = 70$; we can conclude a_4 is accepted. The remaining space would be $10 - s_4 = 7$. We should check $V[3, 7]$ and repeat; Accepted items are a_2 and a_4.



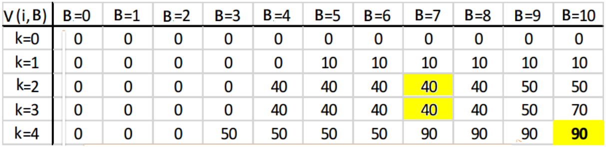
V(i,B)	B=0	B=1	B=2	B=3	B=4	B=5	B=6	B=7	B=8	B=9	B=10
k=0	0	0	0	0	0	0	0	0	0	0	0
k=1	0	0	0	0	0	10	10	10	10	10	10
k=2	0	0	0	0	40	40	40	40	40	50	50
k=3	0	0	0	0	40	40	40	40	40	50	70
k=4	0	0	0	50	50	50	50	90	90	90	**90**

Dynamic Programming for Knapsack

- **Step 4:** Go backwards in the table to retrieve the accepted items.

```
KnapsackRetrieve ($s[], V, n, S$)
    $B \leftarrow S$
    $k \leftarrow n$
    for $k > 0$
        if $V(k, B) = V(k-1, B)$
            report item $k$ as rejected
            $k \leftarrow k - 1$
        else
            report item $k$ as accepted
            $k \leftarrow k - 1$
            $B \leftarrow B - S[k]$
```

  - Here, $S = 10$; sizes are $(5, 4, 6, 3)$ and values are $(10, 40, 30, 50)$.
  - First, $V[4, 10] = 90$ and $V[3, 10] = 70$; we can conclude a_4 is accepted. The remaining space would be $10 - s_4 = 7$. We should check $V[3, 7]$ and repeat; Accepted items are a_2 and a_4.



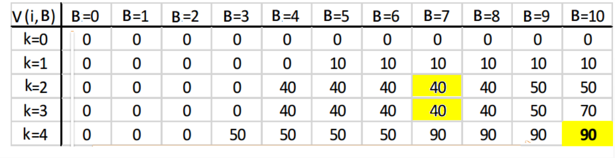
V(i,B)	B=0	B=1	B=2	B=3	B=4	B=5	B=6	B=7	B=8	B=9	B=10
k=0	0	0	0	0	0	0	0	0	0	0	0
k=1	0	0	0	0	0	10	10	10	10	10	10
k=2	0	0	0	0	40	40	40	40	40	50	50
k=3	0	0	0	0	40	40	40	40	40	50	70
k=4	0	0	0	50	50	50	50	90	90	90	**90**

Dynamic Programming for Knapsack

- **Step 4:** Go backwards in the table to retrieve the accepted items.

```
KnapsackRetrieve ($s[], V, n, S$)
    $B \leftarrow S$
    $k \leftarrow n$
    for $k > 0$
        if $V(k,B) = V(k-1,B)$
            report item $k$ as rejected
            $k \leftarrow k-1$
        else
            report item $k$ as accepted
            $k \leftarrow k-1$
            $B \leftarrow B - S[k]$
```

- Here, $S = 10$; sizes are $(5, 4, 6, 3)$ and values are $(10, 40, 30, 50)$.
- First, $V[4, 10] = 90$ and $V[3, 10] = 70$; we can conclude a_4 is accepted. The remaining space would be $10 - s_4 = 7$. We should check $V[3, 7]$ and repeat; Accepted items are a_2 and a_4.



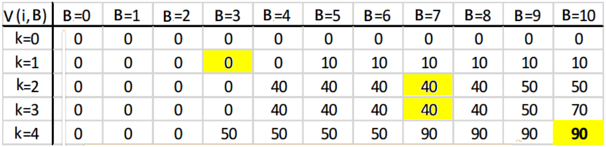
V(i,B)	B=0	B=1	B=2	B=3	B=4	B=5	B=6	B=7	B=8	B=9	B=10
k=0	0	0	0	0	0	0	0	0	0	0	0
k=1	0	0	0	0	0	10	10	10	10	10	10
k=2	0	0	0	0	40	40	40	40	40	50	50
k=3	0	0	0	0	40	40	40	40	40	50	70
k=4	0	0	0	50	50	50	50	90	90	90	**90**

Dynamic Programming for Knapsack

- **Step 4:** Go backwards in the table to retrieve the accepted items.

```
KnapsackRetrieve ($s[], V, n, S$)
    $B \leftarrow S$
    $k \leftarrow n$
    for $k > 0$
        if $V(k,B) = V(k-1,B)$
            report item $k$ as rejected
            $k \leftarrow k-1$
        else
            report item $k$ as accepted
            $k \leftarrow k-1$
            $B \leftarrow B - S[k]$
```

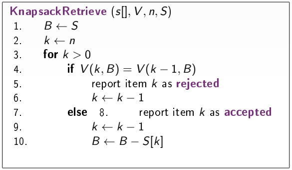
- Here, $S = 10$; sizes are $(5, 4, 6, 3)$ and values are $(10, 40, 30, 50)$.
- First, $V[4, 10] = 90$ and $V[3, 10] = 70$; we can conclude a_4 is accepted. The remaining space would be $10 - s_4 = 7$. We should check $V[3, 7]$ and repeat; Accepted items are a_2 and a_4.



V(i,B)	B=0	B=1	B=2	B=3	B=4	B=5	B=6	B=7	B=8	B=9	B=10
k=0	0	0	0	0	0	0	0	0	0	0	0
k=1	0	0	0	0	0	10	10	10	10	10	10
k=2	0	0	0	0	40	40	40	40	40	50	50
k=3	0	0	0	0	40	40	40	40	40	50	70
k=4	0	0	0	50	50	50	50	90	90	90	**90**

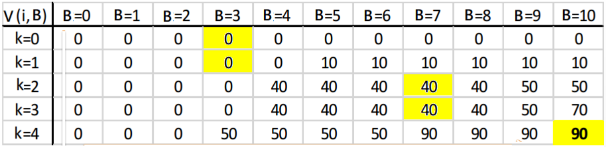
Dynamic Programming for Knapsack

- **Step 4:** Go backwards in the table to retrieve the accepted items.



```
KnapsackRetrieve ($s[], V, n, S$)
    $B \leftarrow S$
    $k \leftarrow n$
    for $k > 0$
        if $V(k, B) = V(k-1, B)$
            report item $k$ as rejected
            $k \leftarrow k - 1$
        else
            report item $k$ as accepted
            $k \leftarrow k - 1$
            $B \leftarrow B - S[k]$
```

  - Here, $S = 10$; sizes are $(5, 4, 6, 3)$ and values are $(10, 40, 30, 50)$.
  - First, $V[4, 10] = 90$ and $V[3, 10] = 70$; we can conclude a_4 is accepted. The remaining space would be $10 - s_4 = 7$. We should check $V[3, 7]$ and repeat; Accepted items are a_2 and a_4.



V(i,B)	B=0	B=1	B=2	B=3	B=4	B=5	B=6	B=7	B=8	B=9	B=10
k=0	0	0	0	0	0	0	0	0	0	0	0
k=1	0	0	0	0	0	10	10	10	10	10	10
k=2	0	0	0	0	40	40	40	40	40	50	50
k=3	0	0	0	0	40	40	40	40	40	50	70
k=4	0	0	0	50	50	50	50	90	90	90	**90**

Matrix Chain Multiplication

- Given a sequence of matrices $A_1, A_2, A_3, \ldots, A_n$, find the best way (using the minimal number of multiplications) to compute their product.
    - Isn't there only one way?$((\ldots((A_1.A_2).A_3)\ldots).A_n)$

Matrix Chain Multiplication

- Given a sequence of matrices $A_1, A_2, A_3, \ldots, A_n$, find the best way (using the minimal number of multiplications) to compute their product.
  - Isn't there only one way?$((\ldots((A_1.A_2).A_3)\ldots).A_n)$
  - No, matrix multiplication is **associative**. e.g. $A_1.(A_2.(A3.(\ldots(A_{n-1}.A_n)\ldots)))$ yields the same matrix.
  - Different multiplication orders do not cost the same:
    - Multiplying p×q matrix A and $q \times r$ matrix B takes $p.q.r$ multiplications; result is a $p \times r$ matrix.

Matrix Chain Multiplication

- Given a sequence of matrices $A_1, A_2, A_3, \ldots, A_n$, find the best way (using the minimal number of multiplications) to compute their product.
  - Isn't there only one way?$((\ldots((A_1.A_2).A_3)\ldots).A_n)$
  - No, matrix multiplication is **associative**. e.g. $A_1.(A_2.(A3.(\ldots(A_{n-1}.A_n)\ldots)))$ yields the same matrix.
  - Different multiplication orders do not cost the same:
    - Multiplying p×q matrix A and $q \times r$ matrix B takes $p.q.r$ multiplications; result is a $p \times r$ matrix.
  - Consider multiplying 10×100 matrix A_1 with 100×5 matrix A_2 and 5×50 matrix A_3.
    - $(A_1.A_2).A_3$ takes $10 \,.\, 100 \,.\, 5 + 10 \,.\, 5 \,.\, 50 = 7500$ multiplications.
    - $A_1.(A_2.A_3)$ takes $100 \,.\, 5 \,.\, 50 + 10 \,.\, 50 \,.\, 100 = 75000$ multiplications.

Subproblem Formulation

- **Step 1:** Define sub-problems to state the optimal solution for each sub-problem in terms of optimal solutions for smaller sub-problems
    - In general, let A_i be $p_{i-1} \times p_i$ matrix.

Subproblem Formulation

- **Step 1:** Define sub-problems to state the optimal solution for each sub-problem in terms of optimal solutions for smaller sub-problems
    - In general, let A_i be $p_{i-1} \times p_i$ matrix.
    - Sub-problem (i, j): product of $A_i, A_{i+1}, \ldots, A_j$.
    - Let $m(i, j)$ be minimal number of multiplications needed to compute $A_i \cdot A_{i+1} \cdot \ldots A_j$; we want to compute $m(1, n)$.

Subproblem Formulation

- **Step 1:** Define sub-problems to state the optimal solution for each sub-problem in terms of optimal solutions for smaller sub-problems
  - In general, let A_i be $p_{i-1} \times p_i$ matrix.
  - Sub-problem (i, j): product of $A_i, A_{i+1}, \ldots, A_j$.
  - Let $m(i, j)$ be minimal number of multiplications needed to compute $A_i \cdot A_{i+1} \cdot \ldots A_j$; we want to compute $m(1, n)$.
  - **Observation:** If $(A(B((CD)(EF))))$ is optimal Then $(B((CD)(EF)))$ is optimal as well

Recursive Formulation

- **Step 2:** Denote the value of the optimal solutions for subproblems recursively.

Recursive Formulation

- **Step 2:** Denote the value of the optimal solutions for subproblems recursively.
    - Assume the position of the last product is k, that is, our final multiplication is of the form $(A_{i+1} \cdot A_{i+2} \cdot \ldots A_k) \cdot (A_{k+1} \cdot A_{k+2} \cdot \ldots A_j)$.

Recursive Formulation

- **Step 2:** Denote the value of the optimal solutions for subproblems recursively.
  - Assume the position of the last product is k, that is, our final multiplication is of the form $(A_{i+1} \cdot A_{i+2} \cdot \ldots A_k) \cdot (A_{k+1} \cdot A_{k+2} \cdot \ldots A_j)$.
  - Consider the case multiplying these 4 matrices: $A : 2 \times 4$ $B : 4 \times 2$ $C : 2 \times 3$ $D : 3 \times 1$

Recursive Formulation

- **Step 2:** Denote the value of the optimal solutions for subproblems recursively.
    - Assume the position of the last product is k, that is, our final multiplication is of the form $(A_{i+1} \cdot A_{i+2} \cdot \ldots A_k) \cdot (A_{k+1} \cdot A_{k+2} \cdot \ldots A_j)$.
    - Consider the case multiplying these 4 matrices: $A : 2 \times 4$ $B : 4 \times 2$ $C : 2 \times 3$ $D : 3 \times 1$
        - (A)(BCD): This is a 2×4 multiplied by a 4×1, so $2 \times 4 \times 1 = 8$ multiplications, plus whatever work it will take to multiply (BCD).

Recursive Formulation

- **Step 2:** Denote the value of the optimal solutions for subproblems recursively.
  - Assume the position of the last product is k, that is, our final multiplication is of the form $(A_{i+1} \cdot A_{i+2} \cdot \ldots A_k) \cdot (A_{k+1} \cdot A_{k+2} \cdot \ldots A_j)$.
  - Consider the case multiplying these 4 matrices: $A : 2 \times 4$ $B : 4 \times 2$ $C : 2 \times 3$ $D : 3 \times 1$
    - (A)(BCD): This is a 2×4 multiplied by a 4×1, so $2 \times 4 \times 1 = 8$ multiplications, plus whatever work it will take to multiply (BCD).
    - (AB)(CD): This is a 2×2 multiplied by a 2×1, so $2 \times 2 \times 1 = 4$ multiplications, plus whatever work it will take to multiply (AB) and (CD).

Recursive Formulation

- **Step 2:** Denote the value of the optimal solutions for subproblems recursively.
  - Assume the position of the last product is k, that is, our final multiplication is of the form $(A_{i+1} \cdot A_{i+2} \cdot \ldots A_k) \cdot (A_{k+1} \cdot A_{k+2} \cdot \ldots A_j)$.
  - Consider the case multiplying these 4 matrices: $A : 2 \times 4$ $B : 4 \times 2$ $C : 2 \times 3$ $D : 3 \times 1$
    - (A)(BCD): This is a 2×4 multiplied by a 4×1, so $2 \times 4 \times 1 = 8$ multiplications, plus whatever work it will take to multiply (BCD).
    - (AB)(CD): This is a 2×2 multiplied by a 2×1, so $2 \times 2 \times 1 = 4$ multiplications, plus whatever work it will take to multiply (AB) and (CD).
    - (ABC)(D): This is a 2×3 multiplied by a 3×1, so $2 \times 3 \times 1 = 6$ multiplications, plus whatever work it will take to multiply (ABC).

Recursive Formulation

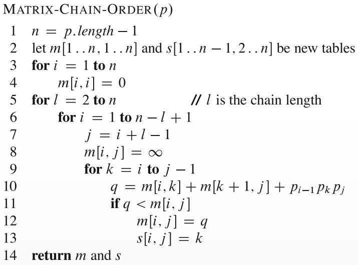
- **Step 2:** Denote the value of the optimal solutions for subproblems recursively.
  - We can compute recursively the best way to multiply the chain from i to k, and from $k+1$ to j, and add the cost of the final product.
  - This means that $m(i,j) = m(i,k) + m(k+1,j) + p_{i-1} \cdot p_k \cdot p_j$
  - Therefore we can write:

$$m(i,j) = \begin{cases} 0 & \text{If } i = j \\ \min_{i \leq k < j}\{m(i,k) + m(k+1,j) + p_{i-1} \cdot p_k \cdot p_j\} & \text{If } i < j \end{cases}$$

Recursive Formulation

- **Step 3:** Fill a dynamic programming table in a bottom-up fashion
- To set $m[i,j]$, we need to look at the values of the same row on the right ($m[i,k]$), or the same column but below ($m[k+1,j]$).

$$m(i,j) = \begin{cases} 0 & \text{If } i = j \\ \min_{i \le k < j}\{m(i,k) + m(k+1,j) + p_{i-1} \cdot p_k \cdot p_j\} & \text{If } i < j \end{cases}$$

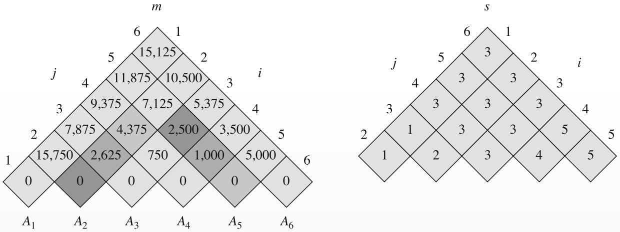


```
MATRIX-CHAIN-ORDER($p$)
$n = p.length - 1$
let $m[1..n, 1..n]$ and $s[1..n-1, 2..n]$ be new tables
for $i = 1$ to $n$
    $m[i,i] = 0$
for $l = 2$ to $n$                // $l$ is the chain length
    for $i = 1$ to $n - l + 1$
        $j = i + l - 1$
        $m[i,j] = \infty$
        for $k = i$ to $j - 1$
            $q = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$
            if $q < m[i,j]$
                $m[i,j] = q$
                $s[i,j] = k$
return $m$ and $s$
```

Recursive Formulation

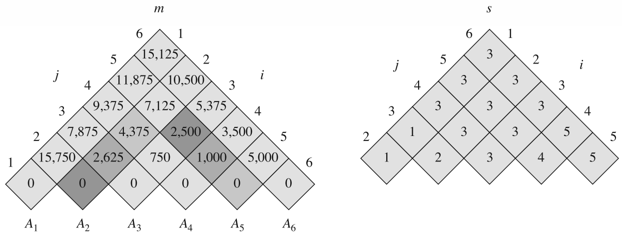
- **Step 3:** Fill a dynamic programming table in a bottom-up fashion
- To set $m[i,j]$, we need to look at the values of the same row on the right ($m[i,k]$), or the same column but below ($m[k+1,j]$).

$$m(i,j) = \begin{cases} 0 & \text{If } i = j \\ \min_{i \le k < j}\{m(i,k) + m(k+1,j) + p_{i-1} \cdot p_k \cdot p_j\} & \text{If } i < j \end{cases}$$



Recursive Formulation

- **Step 3:** Fill a dynamic programming table in a bottom-up fashion



matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

$$\begin{aligned} m[2,5] &= \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13{,}000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11{,}375 \end{cases} \\ &= 7125\,. \end{aligned}$$

Recursive Formulation

- **Step 4:** Retrieve the actual solution using the flag matrix s
- You will work on the details on Assignment 4.

Dynamic Programming Review

1. **Step 1:** define subproblems, and devise the value of the optimal solution for each subproblem using the value of the optimal solutions for smaller subproblems.
2. **Step 2:** write down a recursive formula for the value of optimal solutions.
3. **Step 3:** fill up the dynamic programming table in a bottom-up fashion.
4. **Step 4:** retrieve the actual solution by moving backwards in the table.