# Dynamic Programming[a] (DP)

[a]The term was invited by Richard E. Bellman in the 50s and the term programming refers here to "planning" problems.

# A step back on the divide and conquer approach

## Principle

- **Divide** a problem into independent subproblems
- **Conquer** these subproblems by solving them recursively (or directly when their input size is very small)
- **Combine** the solutions of the subproblems to build the solution of the initial problem

## Example Merge-sort

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

- $A$ is an array of size $n$
- $p, q, r$ indexes between 1 and n
- MERGE is a linear procedure (see next slide)

Example: Apply it and draw the recursive calls on $[5, 2, 6, 1, 5, 2, 3, 1]$

# The Merge-sort algorithm

MERGE procedure: MERGE$(A, p, q, r)$

```
1  n_1 = q − p + 1
2  n_2 = r − q
3  let L[1..n_1 + 1] and R[1..n_2 + 1] be new arrays
4  for i = 1 to n_1
5      L[i] = A[p + i − 1]
6  for j = 1 to n_2
7      R[j] = A[q + j]
8  L[n_1 + 1] = ∞
9  R[n_2 + 1] = ∞
10 i = 1
11 j = 1
12 for k = p to r
13     if L[i] ≤ R[j]
14         A[k] = L[i]
15         i = i + 1
16     else A[k] = R[j]
17         j = j + 1
```

Complexity of the algo: $T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$

$\Rightarrow \Theta(n \log n)$

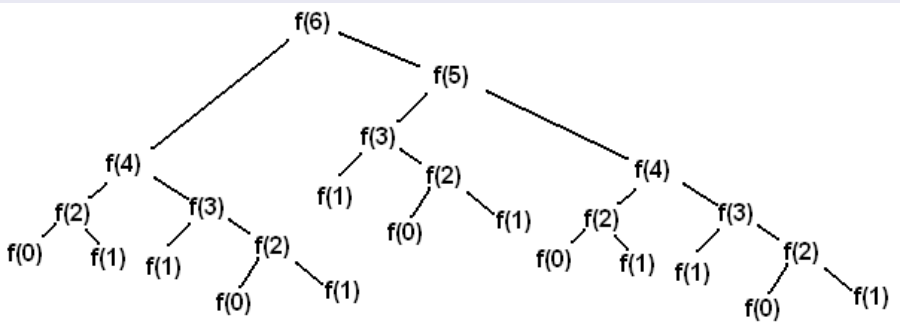
# Another example: the Fibonacci series

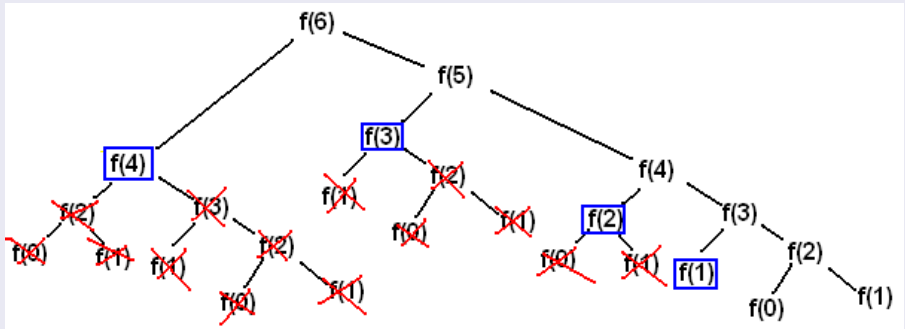
## Computing a Fibonacci number

$$Fib(n) = \begin{cases} \text{if } n = 0 \text{ or } n = 1 \text{ return}(1) \\ \text{otherwise return}(Fib(n-1) + Fib(n-2)) \end{cases}$$

Complexity: $u_n = u_{n-1} + u_{n-2} + 2 \Rightarrow O(1.62^n)$ Exponential!

## Idea: store the results of the calls (memorization): $\Rightarrow O(n)$




$$Fib(n) = \begin{cases} \text{if } n = 0 \text{ or } n = 1\text{: return}(1) \\ \text{if } Fib[n] \neq -1\text{: return}(Fib[n])) \\ \text{otherwise } Fib[n] = Fib[n-1] + Fib[n-2]); \text{return}(Fib[n]) \end{cases}$$

# Dynamic Programming: The Principles

We consider optimization problems a priori difficult to solve. The idea is to solve them by dividing the problem into subproblems and storing the results of each subproblems to make sure that the result is available in O(1) in case of a new call on the same subproblem.

## The 4 steps of a dynamic programming approach

1. Characterize the structure of an optimal solution
2. Recursively define the value associated to an optimal solution according to subproblems
3. Compute the value of an optimal solution by solving the subproblems in a bottom-up fashion (start from the smallest problems and build up solutions to larger and larger problems)
4. Build an optimal solution from computed information

# DP Example 1: matrix chain multiplication

- Find an optimal way to multiply a sequence of $n$ matrices $< A_1, \ldots, A_n >$
- Many solutions are possible: e.g. for $< A_1, A_2, A_3, A_4 >$: $(A_1(A_2(A_3A_4)))$, $(A_1((A_2A_3)A_4))$, $((A_1A_2)(A_3A_4))$, ...
- Recap on matrix multiplication: $A$ has dimension $p \times q$ and $B$ dimension $q \times r$, then the matrix $C : A \cdot B$ has dimension $p \times r$ and the simple algorithm has a complexity in $\Theta(pqr)$
- The multiplication order has some importance: consider 3 matrices $< A_1, A_2, A_3 >$ of dimensions $10 \times 100$, $100 \times 5$ and $5 \times 50$
  - $((A_1A_2)A_3)$ costs $10 * 100 * 5 + 10 * 5 * 50 = 7500$ multiplications
  - $(A_1(A_2A_3))$ costs $100 * 5 * 50 + 10 * 100 * 50 = 75000$ multiplications
- Considering all the possible parenthesizations is intractable $\Omega(4^n/n^{3/2})$ (Catalan number) $\Rightarrow$ use DP

# Matrix chain multiplication

## Formalization of the problem

**Input:** a sequence $< A_1, \ldots, A_n >$ of matrices to multiply, for any $i$ $A_i$ has dimension $p_{i-1} \times p_i$

**Output:** parenthesization of the matrix product minimizing the number of multiplications

# Step1: Characterizing an optimal substructure

Notations: $A_{i\ldots j}$ is the matrix solution of the product $A_i \times A_{i+1} \times \ldots \times A_j$ and $P(i\ldots j)$ be a parenthesization defined on this matrix sequence.

## Consider an optimal parenthesization $P_{opt}(1\ldots n)$ of the sequence

- It divides the product into 2 subproducts at an index $k$:
  $A_1 \times \ldots \times A_n = (A_1 \times \ldots \times A_k) \times (A_{k+1} \times \ldots \times A_n)$
  cost=cost of $A_{1\ldots k}$+cost of $A_{k+1\ldots n}$+cost of $A_{1\ldots k} \times A_{k+1\ldots n}$
- Then $P(1\ldots k)$ the parenthesization defined by $P_{opt}(1\ldots n)$ over $< A_1 \ldots, A_k >$ must be optimal for this sequence
  **Proof:** by contradiction, assume there exists a strictly better $P'(1\ldots k)$ for $< A_1, \ldots, A_k >$, then by replacing $P(1\ldots k)$ by $P'(1\ldots k)$ in $P_{opt}(1\ldots n)$ we obtain a better solution, which contradicts that $P_{opt}(1\ldots n)$ is optimal
  $\Rightarrow P(1\ldots k)$ is optimal
- we can use the same argument for proving that $P(k+1\ldots n)$ is optimal
- $\Rightarrow$ the optimal solution can be decomposed into optimal solutions of subproblems

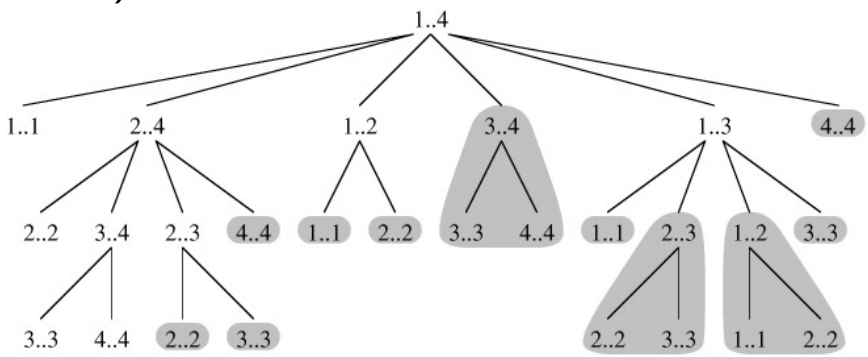
# Step2: computing recursively the optimal solution

## Optimal cost for $A_{i\ldots j}$ $1 \leq i \leq j < n$ stored in $m[i,j]$

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

  $m[1,n]$ is thus the optimal parenthesization cost for the whole sequence
- We also store the optimal $k$ in $s[i,j]$ to be able to build the solution later (ie $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$)

What are the number of subproblems: one for all $i,j$, $1 \leq i \leq j < n$ i.e. $\binom{n}{2} + n = \frac{n(n-1)}{2} + n : \Theta(n^2)$. Note the superposed subproblems in the recursive calls (good indication for DP)



# Step3: algorithm (bottom-up)

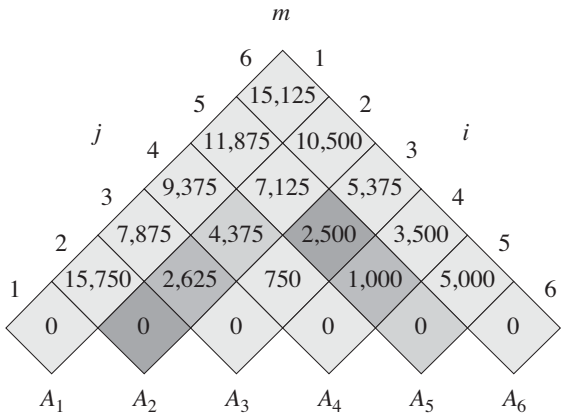
Principle: work on matrix subsequences with increasing length

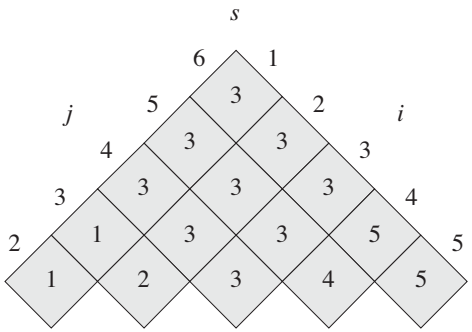
## Algo - complexity $\Theta(n^3)$ (and $\Theta(n^2)$ for storage)

MATRIX-CHAIN-ORDER$(p)$

```
1   n = p.length − 1
2   let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
3   for i = 1 to n
4       m[i, i] = 0
5   for l = 2 to n                    // l is the chain length
6       for i = 1 to n − l + 1
7           j = i + l − 1
8           m[i, j] = ∞
9           for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```

# Example with 6 matrices $< A_1, \ldots, A_6 >$




| matrix | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ | $A_6$ |
|---|---|---|---|---|---|---|
| dimension | $30 \times 35$ | $35 \times 15$ | $15 \times 5$ | $5 \times 10$ | $10 \times 20$ | $20 \times 25$ |

In the this slide, tables are rotated such that the diagonal appears horizontally. The $m$ table uses the main diagonal and upper triangle and the $s$ table only the upper triangle. The optimal solution of the problem is $m[1,6] = 15,125$. Among the darker entries in the previous slide, the pairs with the same shading are taken together in line 10 when computing:

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

# Step4: Constructing an optimal solution

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i == j
2       print "A"_i
3  else print "("
4       PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5       PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6       print ")"
```

This algorithm applied on the previous problem outputs:

$$((A_1(A_2A_3))((A_4A_5)A_6))$$

# Another Example: Longest Common Subsequence (LCS)

## Definitions (sequence, subsequence)

- sequence of letters $X =< x_1, x_2, \ldots, x_n >$
- prefix $X_i =< x_1, \ldots, x_i >$ of size $i$ of $X$ ($X_0$ empty sequence)
- subsequence: original sequence without some of the letters
- common subsequence between two sequences $X$ and $Y$: subsequence both of $X$ and $Y$
- LCS: the longest common subsequence between $X$ and $Y$.
- Example
  - $Z =< A, C, G >$ is not a subsequence of $X < A, G, G, T, C, A >$, but $X' =< A, G >$ is a subsequence of X
  - if $Y =< T, C, A, G, A, T, A >$ then $Z' =< A, G, T, A >$ is a LCS of $X$ and $Y$.
- Applications: DNA sequence alignment, text comparisons, dictionary search

# The problem

## Longest Common Subsequence (LCS)

Input: $X =< x_1, \ldots, x_m >$ and $Y =< y_1, \ldots, y_n >$ 2 sequences

Output: $Z$ a LCS of $X$ and $Y$

# Step1: characterize an optimal solution

**Theorem: Let $Z =< z_1, \ldots, z_k >$ an LCS of $X$ and $Y$**

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and $Z_{k-1}$ is an LCS of $X_{m-1}$ and $Y_{n-1}$.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies $Z$ is an LCS of $X_{m-1}$ and $Y_n$.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies $Z$ is an LCS of $X_m$ and $Y_{n-1}$.

**Proof:** (1) If $z_k \neq x_m$, then we could append $x_m = y_n$ to $Z$ to obtain a common subsequence of $X$ and $Y$ of length $k+1$, contradicting the supposition that $Z$ is a longest common subsequence of $X$ and $Y$. Thus, we must have $z_k = x_m = y_n$. Now, the prefix $Z_{k-1}$ is a length-$(k-1)$ common subsequence of $X_{m-1}$ and $Y_{n-1}$. We wish to show if it is an LCS. Suppose for the purpose of contradiction that there exists a common subsequence $W$ of $X_{m-1}$ and $Y_{n-1}$ with length greater than $k-1$. Then, appending $x_m = y_n$ to $W$ produces a common subsequence of $X$ and $Y$ whose length is greater than $k$, which is a contradiction.

# Step1: characterize an optimal solution

**Theorem: Let $Z =< z_1, \ldots, z_k >$ an LCS of $X$ and $Y$**

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and $Z_{k-1}$ is an LCS of $X_{m-1}$ and $Y_{n-1}$.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies $Z$ is an LCS of $X_{m-1}$ and $Y_n$.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies $Z$ is an LCS of $X_m$ and $Y_{n-1}$.

**Proof (ctd):**
(2) If $z_k \neq x_m$, then $Z$ is a common subsequence of $X_{m-1}$ and $Y$. If there was a common subsequence $W$ of $X_{m-1}$ and $Y_n$ with length greater than $k$, then $W$ would also be a common subsequence of $X_m$ and $Y_n$, contradicting the assumption that $Z$ is an LCS of $X$ and $Y$.
(3) The proof is symmetric to (2). □

- $\Rightarrow$ An LCS of two sequences contains within it an LCS of prefixes of the two sequences
- $\Rightarrow$ LCS has an optimal-substructure property
- $\Rightarrow$ We need then to define a recursive solution with overlapping subproblems.

# Step 2: A recursive solution

- We need to take into account the 3 parts of the previous theorem and take into account limit cases (empty strings).
- Idea: Let $c[i,j]$ be the length of an LCS of the prefix sequences $X_i$ and $Y_j$.
  We can deduce the following recursive formula:

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

$\Rightarrow$ The considered subproblems are based on properties of the original problem (some of them are ruled out by these conditions).

# Step 3: algorithm

$c[i,j]$ table storing the LCS between prefixes, we use an auxiliary table to $b[i,j]$ to build an optimal solution, the arrows indicate which preceding subproblem is used to compute the current solution.

LCS-LENGTH$(X, Y)$

```
1  m = X.length
2  n = Y.length
3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
4  for i = 1 to m
5      c[i, 0] = 0
6  for j = 0 to n
7      c[0, j] = 0
8  for i = 1 to m
9      for j = 1 to n
10         if x_i == y_j
11             c[i, j] = c[i − 1, j − 1] + 1
12             b[i, j] = "↖"
13         elseif c[i − 1, j] ≥ c[i, j − 1]
14             c[i, j] = c[i − 1, j]
15             b[i, j] = "↑"
16         else c[i, j] = c[i, j − 1]
17             b[i, j] = "←"
18 return c and b
```

# Example

Compute the LCS between $X =< A, B, C, B, D, A, B >$ and $Y =< B, D, C, A, B, A >$.

## Tables $b$ and $c$ produced

| $i$ | | $j$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|---|
| | | $y_j$ | | $B$ | $D$ | $C$ | $A$ | $B$ | $A$ |
| 0 | $x_i$ | | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | $A$ | | 0 | ↑ 0 | ↑ 0 | ↑ 0 | ↖ 1 | ← 1 | ↖ 1 |
| 2 | $B$ | | 0 | ↖ 1 | ← 1 | ← 1 | ↑ 1 | ↖ 2 | ← 2 |
| 3 | $C$ | | 0 | ↑ 1 | ↑ 1 | ↖ 2 | ← 2 | ↑ 2 | ↑ 2 |
| 4 | $B$ | | 0 | ↖ 1 | ↑ 1 | ↑ 2 | ↑ 2 | ↖ 3 | ← 3 |
| 5 | $D$ | | 0 | ↑ 1 | ↖ 2 | ↑ 2 | ↑ 2 | ↑ 3 | ↑ 3 |
| 6 | $A$ | | 0 | ↑ 1 | ↑ 2 | ↑ 2 | ↖ 3 | ↑ 3 | ↖ 4 |
| 7 | $B$ | | 0 | ↖ 1 | ↑ 2 | ↑ 2 | ↑ 3 | ↖ 4 | ↑ 4 |

One possible LCS is $< B, C, B, A >$ of length 4, the shaded entries illustrate how the sequence is obtained. Is there another solution?

# Step 4: building an optimal solution

We use the matrix $b$, the initial call is PRINT-LCS($b, X, X.length, Y.length$).

```
PRINT-LCS(b, X, i, j)
1  if i == 0 or j == 0
2      return
3  if b[i, j] == "↖"
4      PRINT-LCS(b, X, i − 1, j − 1)
5      print x_i
6  elseif b[i, j] == "↑"
7      PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```

## Complexity analysis

- The algo LCS-LENGTH is $\Theta(mn)$ (each entry of the $c$ tab is computed in $O(1)$)
- The PRINT-LCS procedure is $O(m + n)$

# Dynamic programming - Summary and Take home message

To solve optimization problems by dynamic prog., 2 properties are necessary:

- Optimal substructure property: an optimal solution can be decomposed into optimal solutions of sub-problems. This property can be shown by using proofs by contradiction.
  Sometimes, there exist many ways to arrange the solution of a problem according to subproblems, the best solution is the one leading to the lowest complexity. There exists no general principle for doing it, "just" try to have a natural expression of the subproblem ;-).
- Overlapping subproblems property: the subproblems must overlap, *i.e.* be called many times to solve the original problem. If this is not the case, a divide and conquer approach is more relevant.
- In the examples we saw, we used a bottom-up approach which implies in fact that we are able to define "the bottom" - *i.e.* where to begin the algorithm. Sometimes it is not possible or not easy (chess game for example), in this case we use a top-down procedure by starting from the big original problem and call the subproblems: we check if a subproblem has already been solved, if it's not the case you solve it recursively otherwise you take the available solution from a (relevant) table.
- We can also solve problems that are not optimization pbs (*e.g.* Fibonacci).

# A remark on the optimality principle

## DP relies on the Optimality principle from Richard Bellman

*Every sub-policy of an optimal policy is also optimal.*
For applying dynamic programming we take the hypothesis that:
*The optimal solution of a problem can be obtained from the optimal solutions of the associated sub-problems.*

Even if the proofs made in these lectures may appear "trivial", this is not always possible/true.
Let us consider two different problems of path search in an unweighed oriented graph:

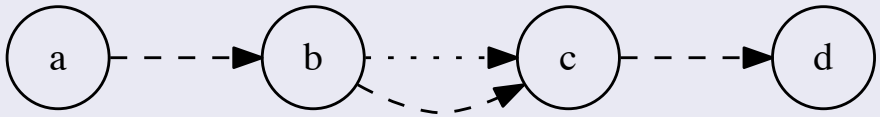
- Shortest path problem $\rightarrow$ the optimality principle holds
- Longest path problem $\rightarrow$ the optimality principle **does not hold**

# Shortest path in a oriented graph

- Consider an unweighted oriented graph $G$ and the problem of computing the shortest path between 2 nodes.
- **Claim**: A path is of minimal length if all its subpaths are of minimal length

## Proof by contradiction

- Consider a shortest path between 2 nodes a and d (dashed line) that goes threw two other nodes b and c.
  Let $[a - d]$ be the length of the path between a and d on this path.

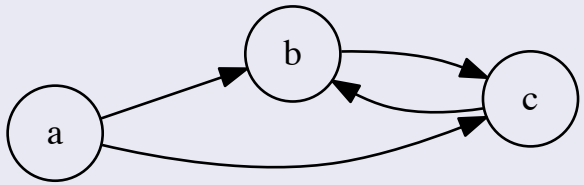



- The length of $[a - d] = [a - b] + [b - c] + [c - d]$.
- Suppose there exists one subpath that is not optimal, *e.g.* $[b - c]$
- Then there exists another subpath between b and c of smaller size (dotted line), by using this path between b and c, we can obtain a path shorter path than $[a - d]$ which leads to a contradiction with respect to the optimality of $[a - d]$.

$\Rightarrow$ Classic proof

# Longest path in an oriented graph (without loops)

- Consider an unweighted oriented graph $G$ with loops and the problem of computing the longest path between 2 nodes.
- Consider the graph below and two nodes a and c




- The longest path (without loop) between a and c is of length 2 ($a \rightarrow b \rightarrow c$), the longest path between a and b is also of length 2 ($a \rightarrow c \rightarrow b$), and the longest one between b and c is of length 1.
- The longest path between a and c can then not be obtained by the composition of longest subpaths since the longest path between a and b is of size 2 !
- The optimality principle is not valid in this case.

Note: this does not mean that applying dynamic programming is totally impossible, because one could imagine another strategy for modeling the problem. However, this may be a good indicator that applying dynamic programming is at least difficult (or even impossible) and then we may consider other approaches such as branch and bound.