

Dynamic Programming

IN3130

INF3130: Dynamic Programming

- In the textbook: Ch. 9, and Section 20.5
- The slides presented here have a different introduction to this topic than the textbook
 - This is done because the introduction in the textbook seems rather confusing.
 - NB: The formulation of the «principle of optimality» (def. 9.1.1) should in fact *be the other way around!*
- And the curriculum in this course is the version used in these slides, not the introduction in the textbook.
- These slides have a lot of text
 - Meant to be a presentation that can be read afterwards
 - This is usually also the style in my slides

Dynamic programming

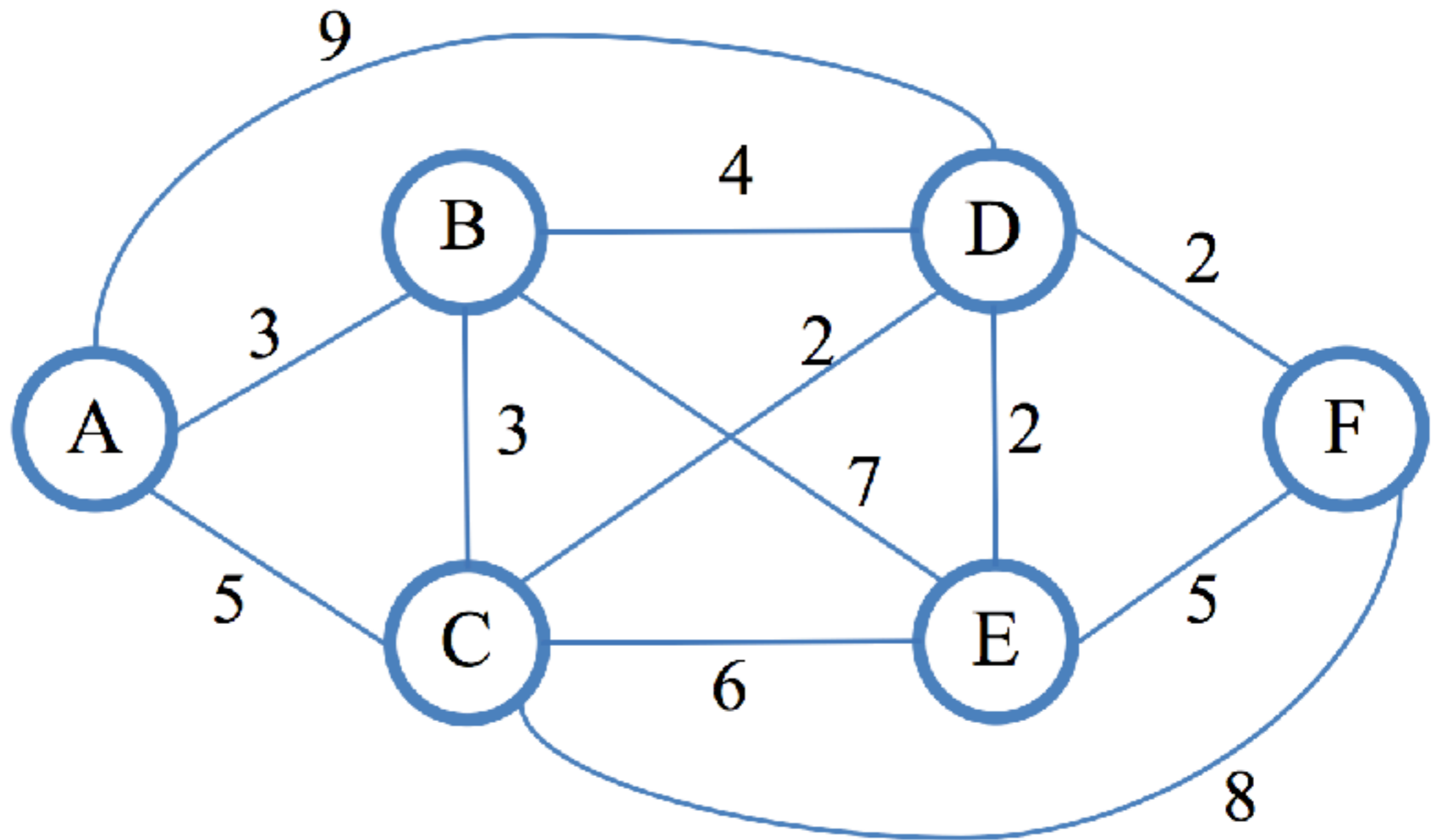
Dynamic programming was formalised by Richard Bellmann (RAND Corporation) in the 1950'es.

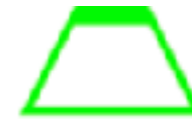
- «programming» should here be understood as planning, or making decisions. It has nothing to do with writing code.
- "*Dynamic*" should indicate that it is a stepwise process.



But was that the real background for the name??

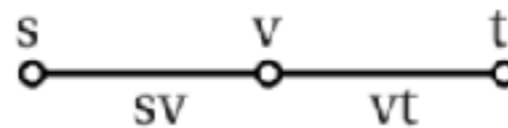
Ex. 1: All pairs shortest paths

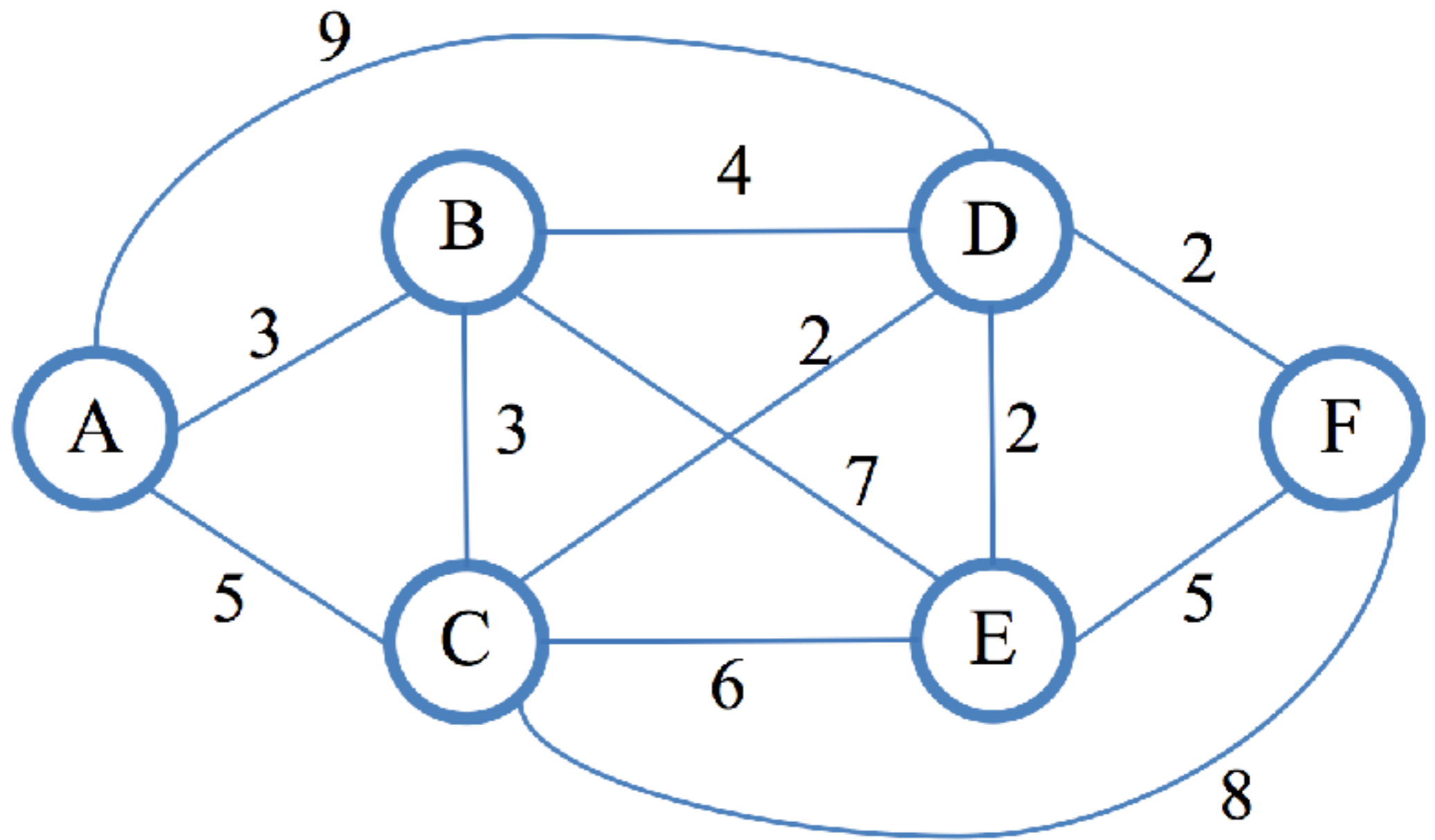




Dynamic Programming

- Building up a solution from solutions from subproblems
- Principle: Every part of an optimal solution must be optimal.





$$d(i,j) \leftarrow \min_k [(d(i,k) + d(k,j)), d(i,j)]$$

Ex. 2: 0-1 Knapsack

In 0-1 KNAPSACK we are given integers w_1, w_2, \dots, w_n and K , and we must decide whether there is a subset S of $\{1, 2, \dots, n\}$ such that $\sum_{j \in S} w_j = K$. In other words: Can we put a subset of the integers into our knapsack such that the knapsack sums up to exactly K , under the restriction that we include any w_i at most one time in the knapsack.

Note: This decision version of 0-1 KNAPSACK is essentially SUBSET SUM.

0-1 KNAPSACK can be solved by dynamic programming. **Idea:** Going through all the w_i one by one, maintain an (ordered) set M of all sums ($\leq K$) which can be computed by using some subset of the integers seen so far.



Algorithm DP

```
1. Let  $M_0 := \{0\}$ .
2. For  $j = 1, 2, \dots, n$  do:
    Let  $M_j := M_{j-1}$ .
    For each element  $u \in M_{j-1}$ :
        Add  $v = w_j + u$  to  $M_j$  if  $v \leq K$  and
         $v$  is not already in  $M_j$ .
3. Answer 'Yes' if  $K \in M_n$ , 'No'
   otherwise.
```

Example: Consider the instance with w_i 's
11, 18, 24, 42, 15, 7 and $K = 56$. We get the
following M_i -sets:

$M_0 : \{0\}$
 $M_1 : \{0, 11\} \quad (0 + 11 = 11)$
 $M_2 : \{0, 11, 18, 29\} \quad (0 + 18 = 18, 11 + 18 = 29)$
 $M_3 : \{0, 11, 18, 24, 29, 35, 42, 53\}$
 $M_4 : \{0, 11, 18, 24, 29, 35, 42, 53\}$
 $M_5 : \{0, 11, 15, 18, 24, 26, 29, 33,$
 $35, 39, 42, 44, 50, 53\}$
 $M_6 : \{0, 7, 11, 15, 18, 22, 24, 25, 26, 29, 31, 33,$
 $35, 36, 39, 40, 42, 44, 46, 49, 50, 51, 53\}$

Theorem 1 *DP is a pseudo-polynomial
algorithm. The running time of DP is*
 $\mathcal{O}(nK \log K)$.

Proof: MAXINT(I) = $K \dots$

A simple example

We are given a matrix W with positive «weights» in each cell:

W :

12	5	35	7	21
4	29	8	19	14
8	3	19	20	24
37	84	78	15	62
26	13	40	33	12
21	60	27	18	17

Problem: Find the «best» path (lowest sum of weights) from upper left to lower right corner.

NB: The shown red path is randomly chosen, and is probably *not* the best path (has weight = 255)

We use a new matrix P to store intermediate results:

$P[i,j]$ = The weight of the best path from the start (upper left) to cell $[i,j]$.

The «recurrence relation» will be:

$$P[i, j] = \min(P[i-1, j], P[i, j-1]) + W[i, j]$$

- We can initialize by filling in the leftmost column and topmost row, as shown to the left.
- We can fill in P according to the formula above

Questions (exercises for next week):

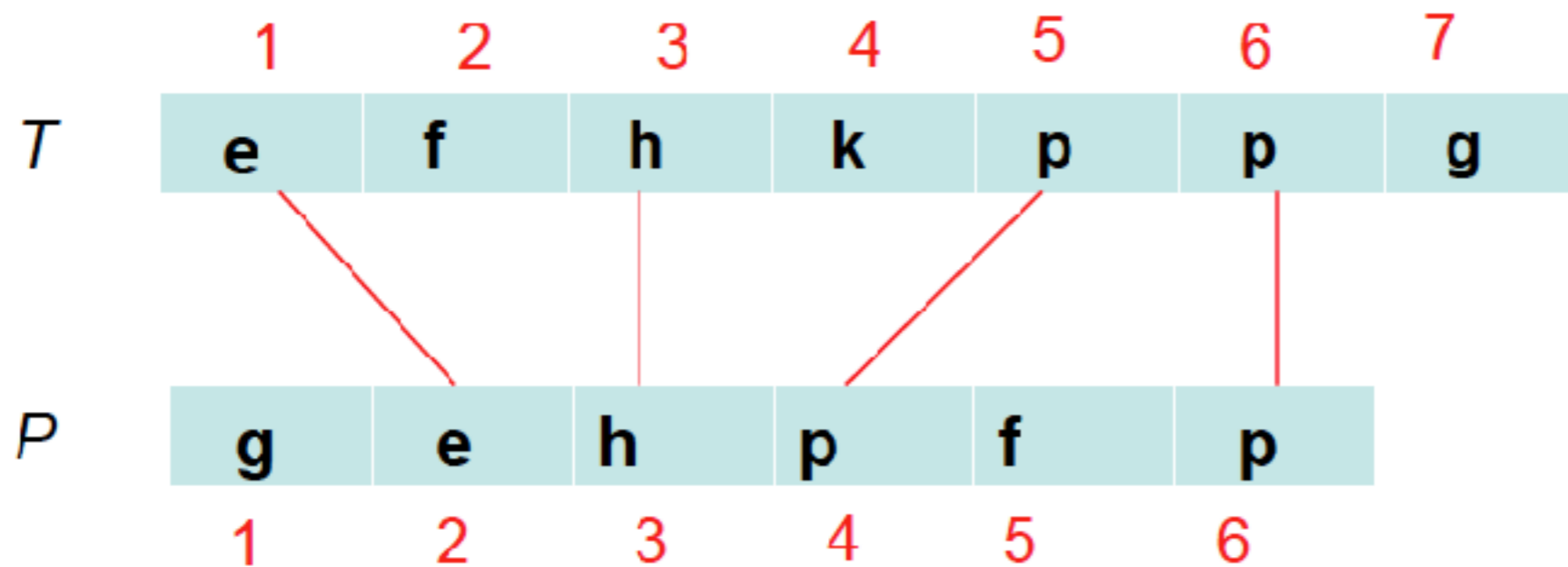
- In which order should P be filled out?
- How can we find the shortest path itself?
- What is the complexity of this algorithm?

P (initialization in black)

12	17	52	59	80
16	45	53	72	86
24	27	46	66	90
61	111	124	81	143
87	100	140	114	126
108	160	167	132	143

Another problem (Ch. 9.4)

Find the «Longest Common Subsequence» of two strings: P (pattern) and T (text)



The Longest Common Subsequence is here

«e, h, p, p»

«Length of Longest Common Subsequence» (LCS)
is 4.

An idea for finding LCS

We will use an integer matrix $L[0:m, 0:n]$ as shown below, where we imagine that the string $P = P[1:m]$ is placed downwards along the left side of L , and $T = T[1:n]$ is placed above L from left to right (at corresponding indices).

(NB: This is a slightly different use of indices than in Sections 9.4 and 20.5).

Our plan is then to systematically fill in this table so that

$$L[i, j] = \text{LSC}(P[1:i], T[1:j])$$

We will do this from «smaller» to «larger» index pairs (i, j) , by taking column after column from left to right (**but row after row from top to bottom would also work**). The value we are looking for, $\text{LSC}(P, T)$, will then occur in $L[m, n]$ when all entries are filled.

The matrix L :

	T →									
	0	1	...	j-1	j					n
P ⁰										
1										
⋮										
i-1										
i					?					
m										

Example: $P = \text{«gehpfp»}$ and $T = \text{«efhkppg»}$

We initialize the leftmost column and the topmost row as below

- Why is it correct with zeroes here?
- Note that these cells correspond to the empty prefix of P and/or the empty prefix of T .

empty prefix of T .

		T														
		<table><tr><td>e</td><td>f</td><td>h</td><td>k</td><td>p</td><td>p</td><td>g</td></tr></table>								e	f	h	k	p	p	g
e	f	h	k	p	p	g										
L		0	1	2	3	4	5	6	7	j						
P	g	0	0	0	0	0	0	0	0							
	e	0	0	0	0	0	0	0	1							
	h	0	1	1	1											
	p	0	1	1	2											
	f	0														
	p	0														
		0							4?							
i																

We have filled in a few more entries of L by intuition

We hope to get 4 here!

The general formula for filling L

		T							
		e	f	h	k	p	p	g	
P	L	0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
	1	0	0	0	0	0	0	0	1
	2	0	1	1	1				
	3	0	1	1	2				
	4	0	1						
	5	0							
	6	0							4?

We want to find: $L[i,j]$
and assume we have
already computed:

$L[i-1,j-1]$	$L[i-1,j]$
$L[i,j-1]$	(Find: $L[i,j]$)

Case 1:

If $P_i = T_j$ then

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

WHY?

Case 2:

If $P_i \neq T_j$ then

$$L[i,j] = \max(L[i,j-1], L[i-1,j])$$

WHY?

$L[i-1, j-1]$	$L[i-1, j]$
$L[i, j-1]$	(Find: $L[i, j]$)

Case 1:

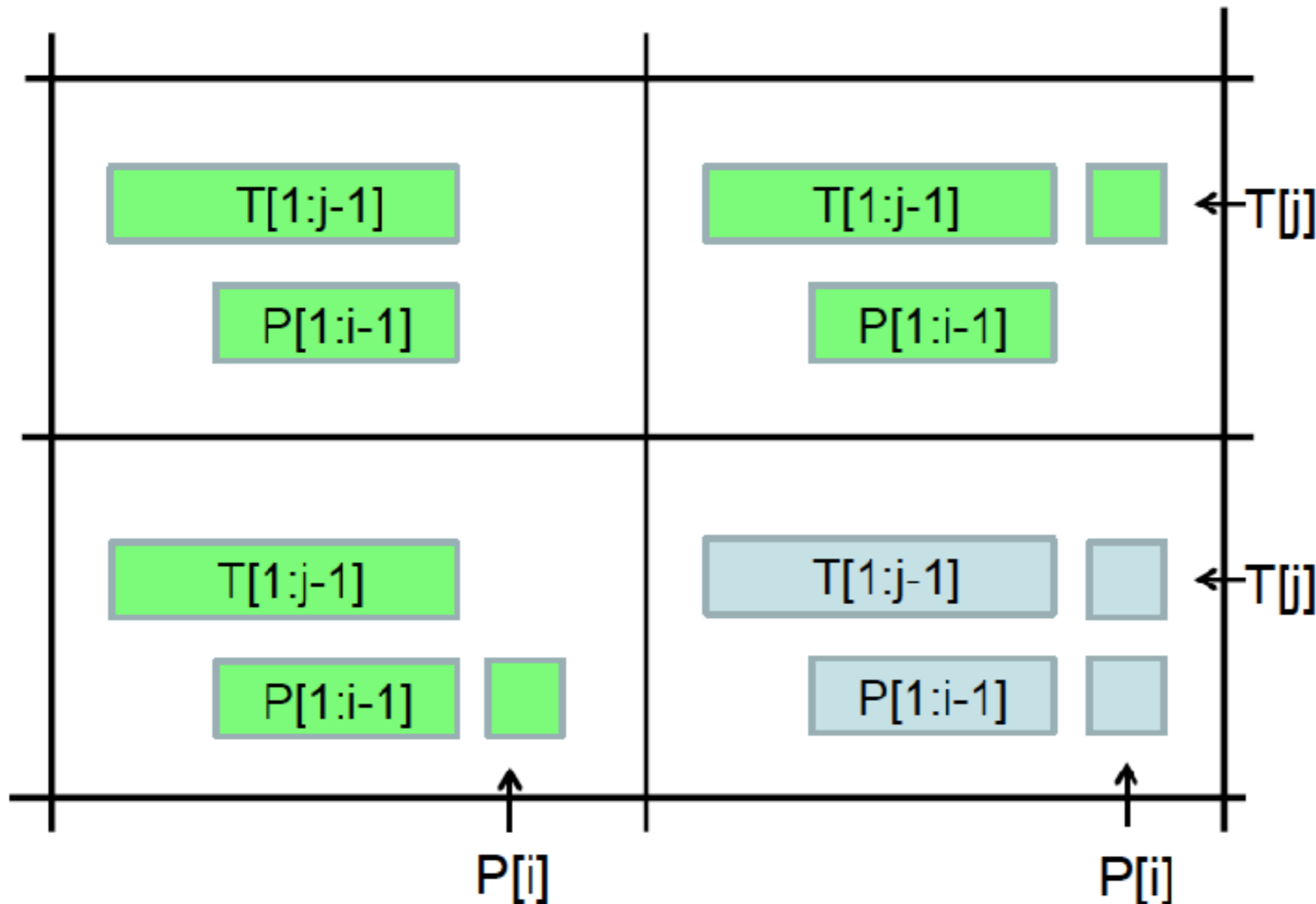
If $P_i = T_j$ then

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

Case 2:

If $P_i \neq T_j$ then

$$L[i, j] = \max(L[i, j-1], L[i-1, j])$$



Using the formula for filling L

		T							
		e	f	h	k	p	p	g	
P	L	0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
	1	0	0	0	0	0	0	0	1
	2	0	1	1	1	1	1	1	1
	3	0	1	1	2	2	2	2	2
	4	0	1	1	2	2	3	3	3
	5	0	1	2	2	2	3	3	3
6	0	1	2	2	2	3	4	4	
		g	e	h	p	f	p		

We want to find: $L[i,j]$
and assume we have
already computed:

$L[i-1,j-1]$

$L[i-1,j]$

$L[i,j-1]$

(Find: $L[i,j]$)

Hurrah!

Case 1:

If $P_i = T_j$ then

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

Case 2:

If $P_i \neq T_j$ then

$$L[i,j] = \max(L[i,j-1], L[i-1,j])$$

Finding the Longest Common Subsequence itself

		T							
		e	f	h	k	p	p	g	
L		0	1	2	3	4	5	6	7
P	g	0	0	0	0	0	0	0	0
	e	0	0	0	0	0	0	0	1
	h	0	1	1	2	2	2	2	2
	p	0	1	1	2	2	3	3	3
	u	0	1	1	2	2	3	3	3
	p	0	1	2	2	2	3	4	4
	p	0	1	2	2	2	3	4	4

Case 1

If $P_i = T_j$ then

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

Case 2

If $P_i \neq T_j$ then

$$L[i,j] = \max(L[i,j-1], L[i-1,j])$$

1. To find the actual Longest Common Subsequence we highlight entries, backwards from lower right, what «caused» each value (green numbers)

2. The red arrows indicate the letters included in the Longest Common Subsequence

We'll now look at the problem discussed
in Chapter 20.5:

«Approximate String Matching»:

Given: A long string T and a shorter string P[?]

Problem: Find strings «similar» to P in T

P: uttxv

T: b s uttv r t o x i g uttv x l b t s k uttzxv k l v h u uttxv n x utztxv w

Questions:

- What do we mean by a «similar string»?
- Can we quantify the degree of similarity?

We'll first look at how to define and find:

The Edit Distance between P and T

The «edit distance» between two strings

We observe that any string P can be converted to another string T by some sequence of the following operations (usually by many different such sequences):



- Substitution:** One symbol in P is changed to another symbol.
- Addition:** A new symbol is inserted somewhere in P .
- Removing:** One symbol is removed from P .

The «Edit Distance», $ED(P, T)$, between two strings P and T is:

The smallest number of such operations needed to convert P to T
(or T to P ! Note that the definition is symmetric in P and T !)

Example.

logarithm \rightarrow alogarithm \rightarrow algarithm \rightarrow algorithm (Steps: +a, -o, a \rightarrow o)
 P T

Thus $ED(\text{"logarithm"}, \text{"algorithm"}) = 3$ (as there are no shorter ways!) ¹²

To find $ED(P, T)$ we use a similar setup as for LCS

We use an integer matrix $D[0:m, 0:n]$, where $P = P[1:m]$ is placed downwards along the left side of D , and $T = T[1:n]$ is placed above D

Our plan is again to systematically fill in this table, but so that

$$D[i, j] = \text{Edit Distance between the strings } P[1:i] \text{ and } T[1:j]$$

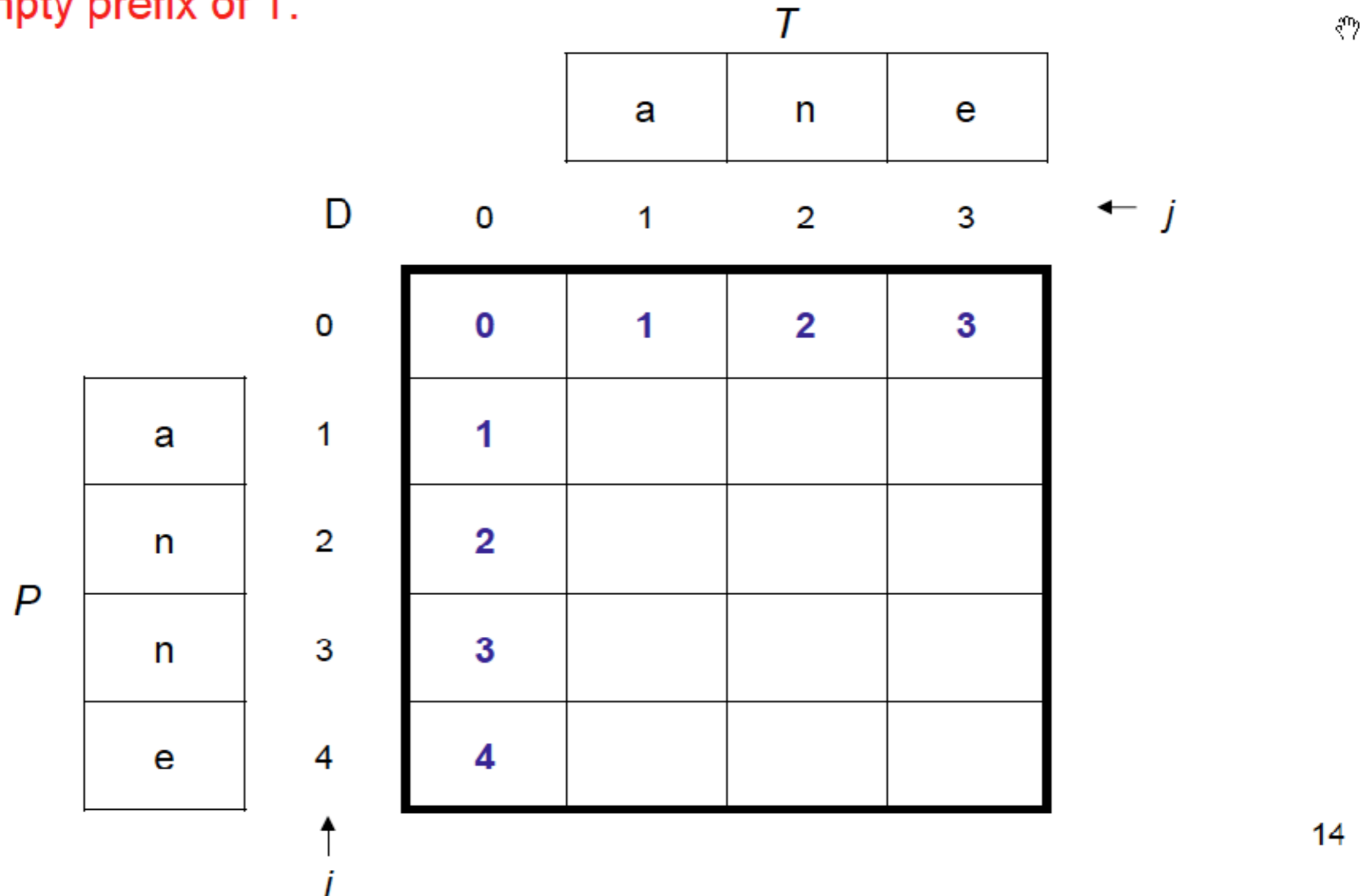
Like before we will do this e.g. by taking column after column from left to right. The value we are looking for, $ED(P, T)$, will then occur in $D[m, n]$ when all entries are filled in.

The matrix D :

		T							
		0	1	...	$j-1$	j			n
0									
1									
\vdots									
$i-1$									
i						?			
m									

Example: $P = \text{«anne»}$ and $T = \text{«ane»}$

- We initialize the leftmost column and the topmost row as below
- Why is this correct?
- Note that these cells correspond to the empty prefix of P and/or the empty prefix of T .

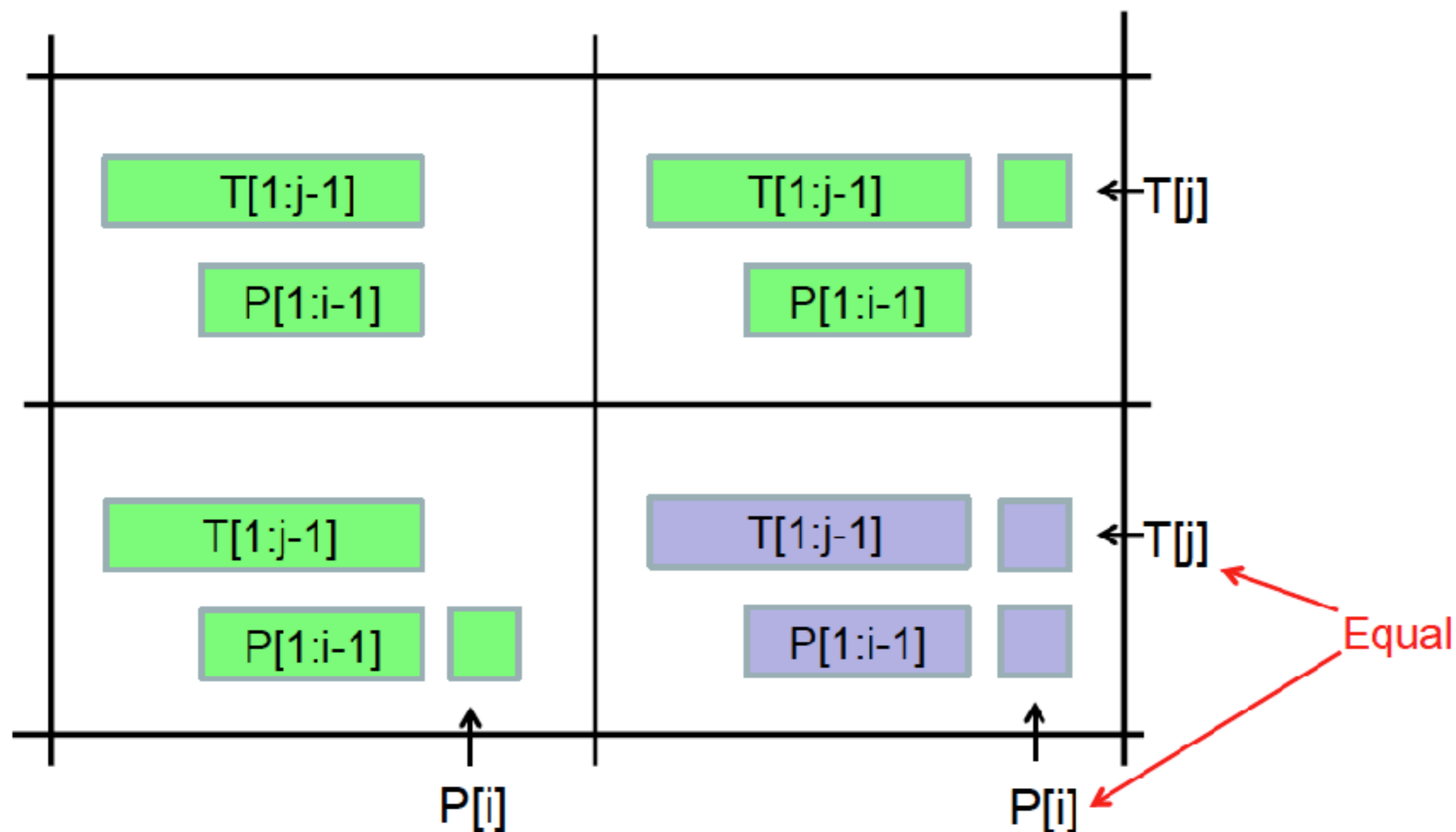


$L[i-1, j-1]$	$L[i-1, j]$
$L[i, j-1]$	(Find: $L[i, j]$)

Case 1:

If $P_i = T_j$ then

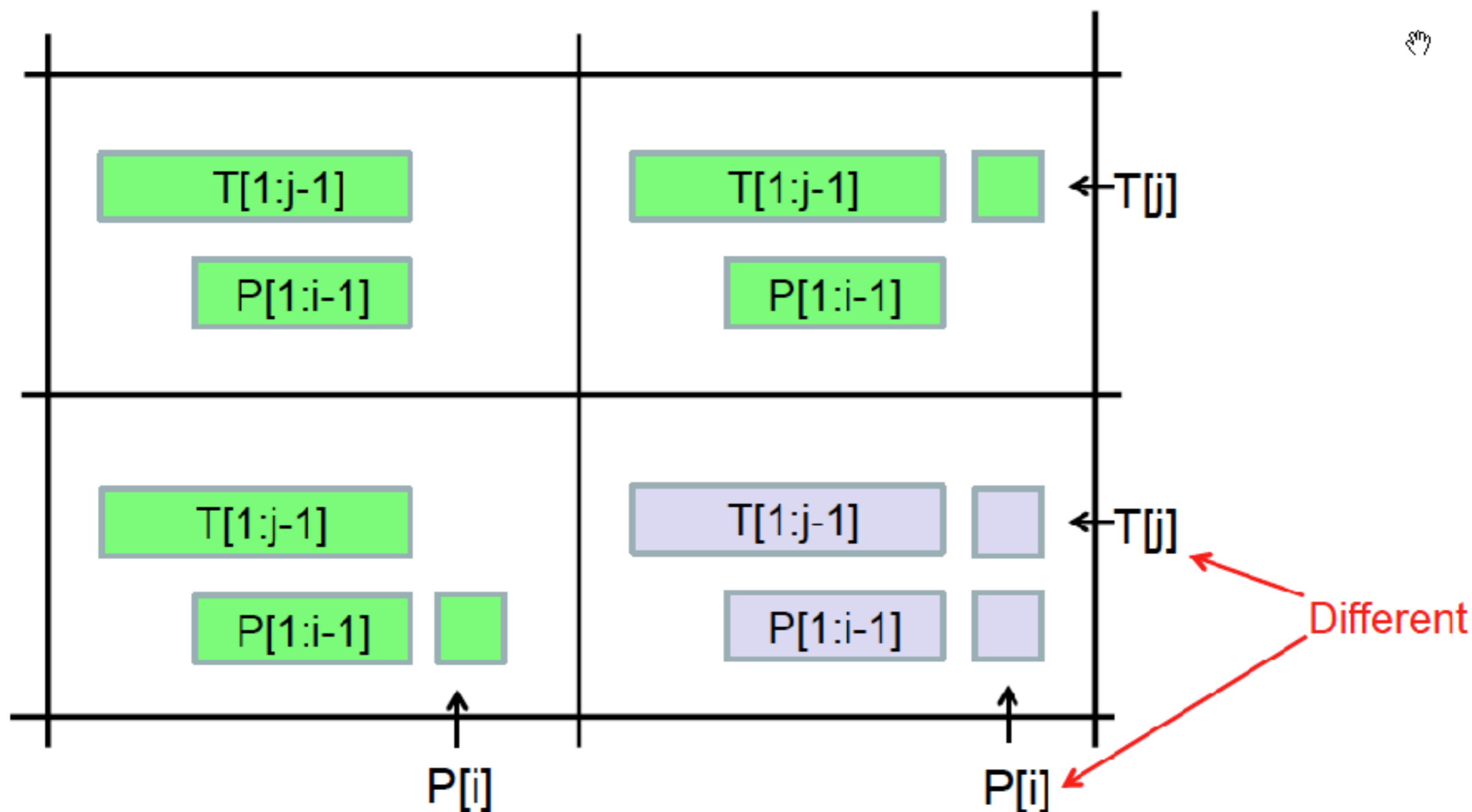
$$D[i, j] = D[i-1, j-1]$$



$L[i-1, j-1]$	$L[i-1, j]$
$L[i, j-1]$	(Find: $L[i, j]$)

Case 2:

If $P_i \neq T_j$ then $D[i, j] =$
 $\min(D[i-1, j-1], D[i, j-1], D[i-1, j]) + 1$



The general recurrence relation becomes

To fill in this matrix D we in fact used the relation indicated below (from Ch 20.5)

$$D[i, j] = \begin{cases} D[i-1, j-1] & \text{hvis } P[i] = T[j] \\ \min \left\{ \underbrace{D[i-1, j-1] + 1}_{\text{substitusjon}}, \underbrace{D[i-1, j] + 1}_{\substack{\text{tillegg i } T \\ \text{sletting i } P}}, \underbrace{D[i, j-1] + 1}_{\text{sletting i } T} \right\} & \text{ellers} \end{cases}$$

$$D[0, 0] = 0, \quad D[i, 0] = D[0, i] = i.$$

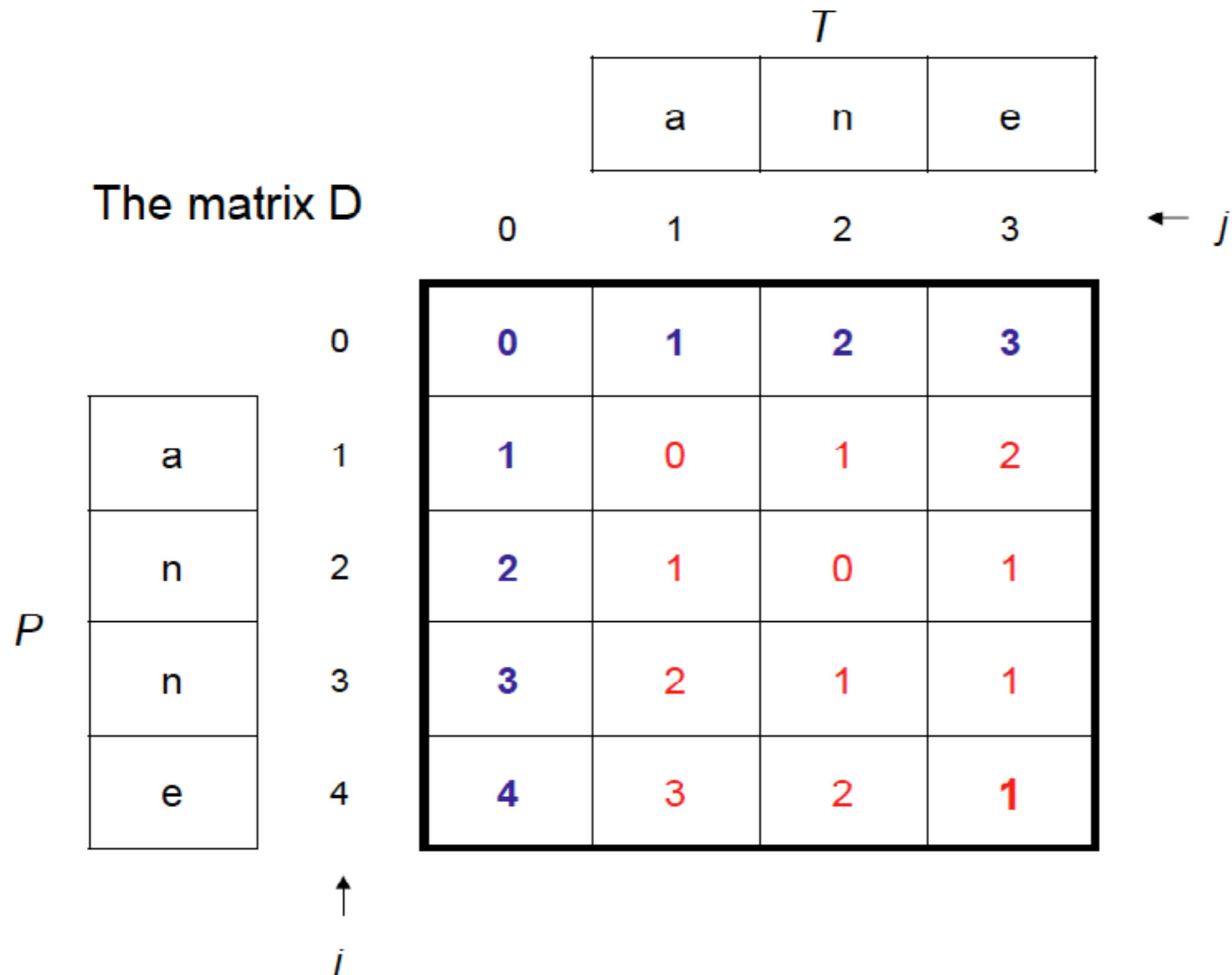
The equalities at the last line can be used to initialize the matrix (shown in red).

The matrix D :

		<div style="display: flex; align-items: center;"> T → </div>							
		0	1	...	j-1	j			n
0		0	1		j-1	j			n
1		1							
⋮									
i-1		i-1							
i					?				
m									

Example: $P = \text{«anne»}$ and $T = \text{«ane»}$

- Using the rules
- The answer $ED(P, T)$ will appear in $D[4, 3]$ (lower right)



A program for computing the edit distance

```
function EditDistance (  $P[1:m]$ ,  $T[1:n]$  )  
  for  $i \leftarrow 0$  to  $m$  do  $D[i, 0] \leftarrow i$  endfor           // Initialize row zero  
  for  $j \leftarrow 1$  to  $n$  do  $D[0, j] \leftarrow j$  endfor       // Initialize column zero  
  
  for  $i \leftarrow 1$  to  $m$  do  
    for  $j \leftarrow 1$  to  $n$  do  
      if  $P[i] = T[j]$  then  
         $D[i, j] \leftarrow D[i-1, j-1]$   
      else  
         $D[i, j] \leftarrow \min(D[i-1, j-1] + 1, D[i-1, j] + 1, D[i, j-1] + 1)$   
      endif  
    endfor  
  endfor  
  return(  $D[m, n]$  )  
end EditDistance
```

Note that, after the initialization, we look at the pairs (i, j) in the following order (line after line):

(1,1) (1,2) ... (1,n)
(2,1) (2,2) ... (2,n)
...
(m,1) ... (m,n)

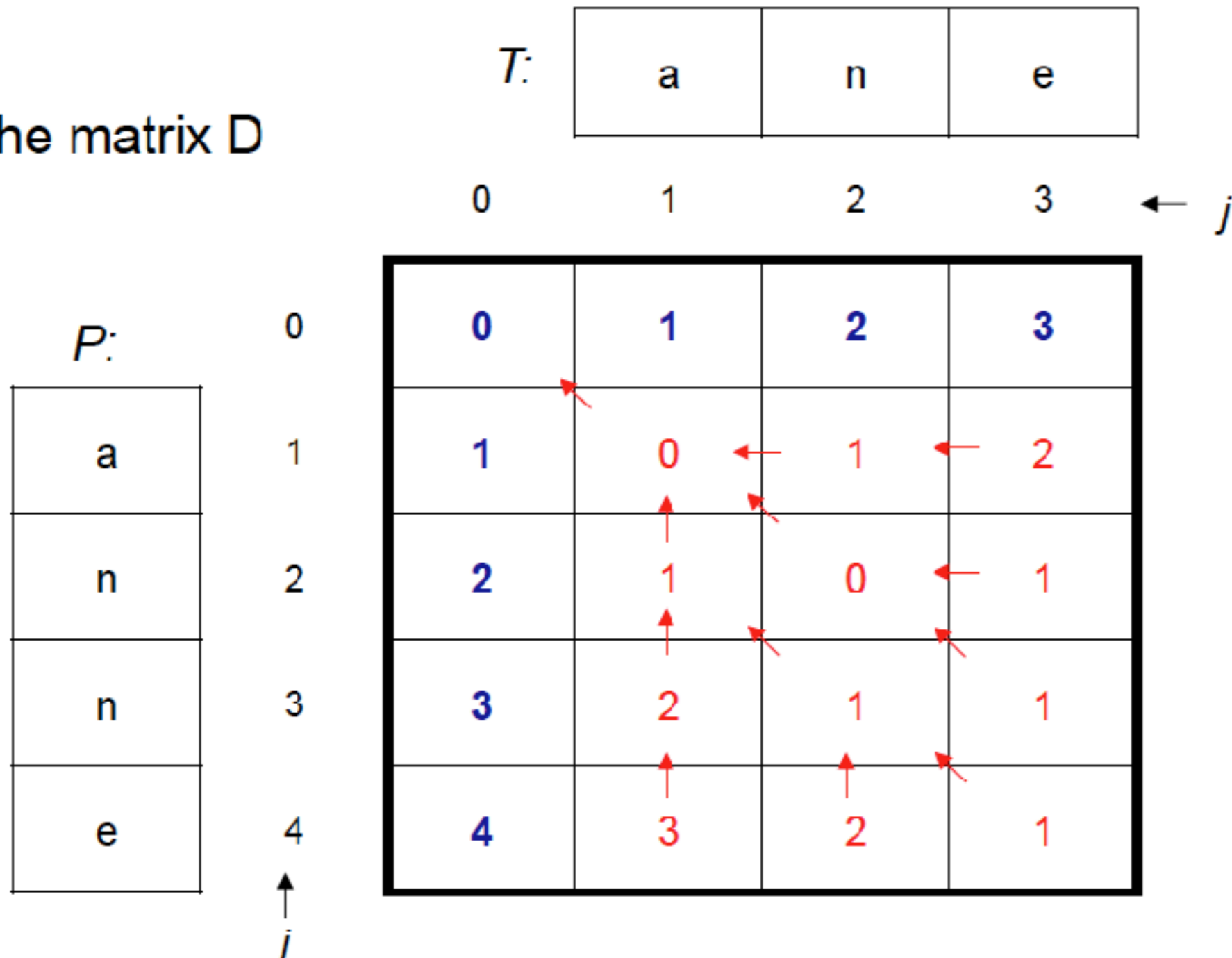
This is OK as this order ensures that the smaller instances are solved before they are needed to solve a larger instance. That is:

$D[i-1, j-1]$, $D[i-1, j]$ and $D[i, j-1]$
are always computed before $D[i, j]$

Our old example: ED(«anne», «ane»)

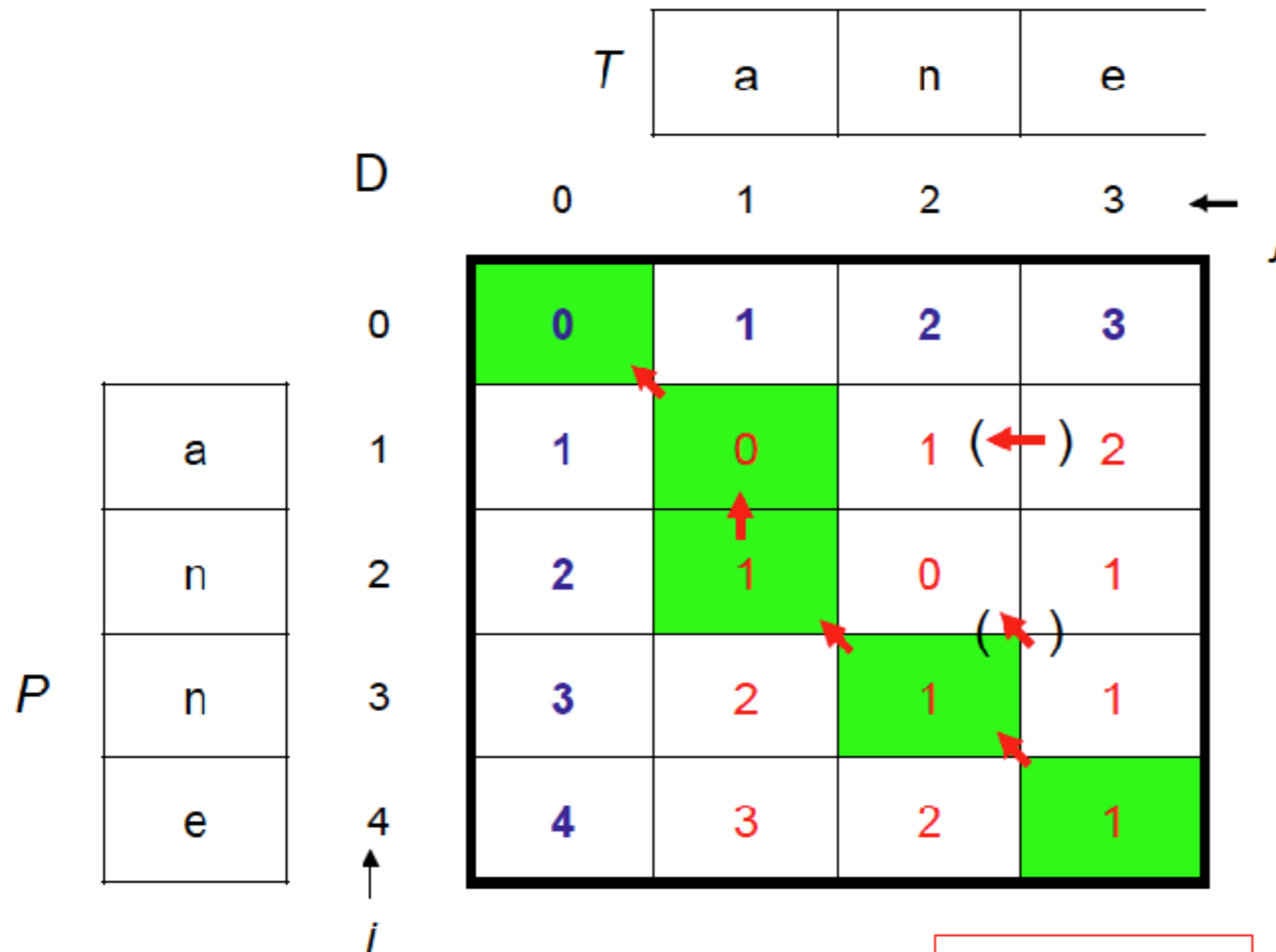
The value used in each entry is given by an arrow into that entry.

The matrix D



Finding the edit steps:

Follow the «path» used from the final entry backwards to [0,0]. The meaning of each step is given to the right, assuming that P is transformed to T .



- ↖ Diagonally, and $P[i] = T[j]$:
- No edit was needed.
Occurred e.g. for $D[3, 2]$.
- ↖ Diagonally, and $P[i] \neq T[j]$:
- Substitution Occurred
e.g. for $D[3, 3]$ (not used in the current final edit path)
- ↑ Upwards (and thus $P[i] \neq T[j]$):
- A letter is deleted from P
Occur e.g. for $D[2, 1]$
- ← Towards the left (and thus $P[i] \neq T[j]$):
A letter is added to P
Occur e.g. for $D[1, 3]$ (not used in the current final edit path)

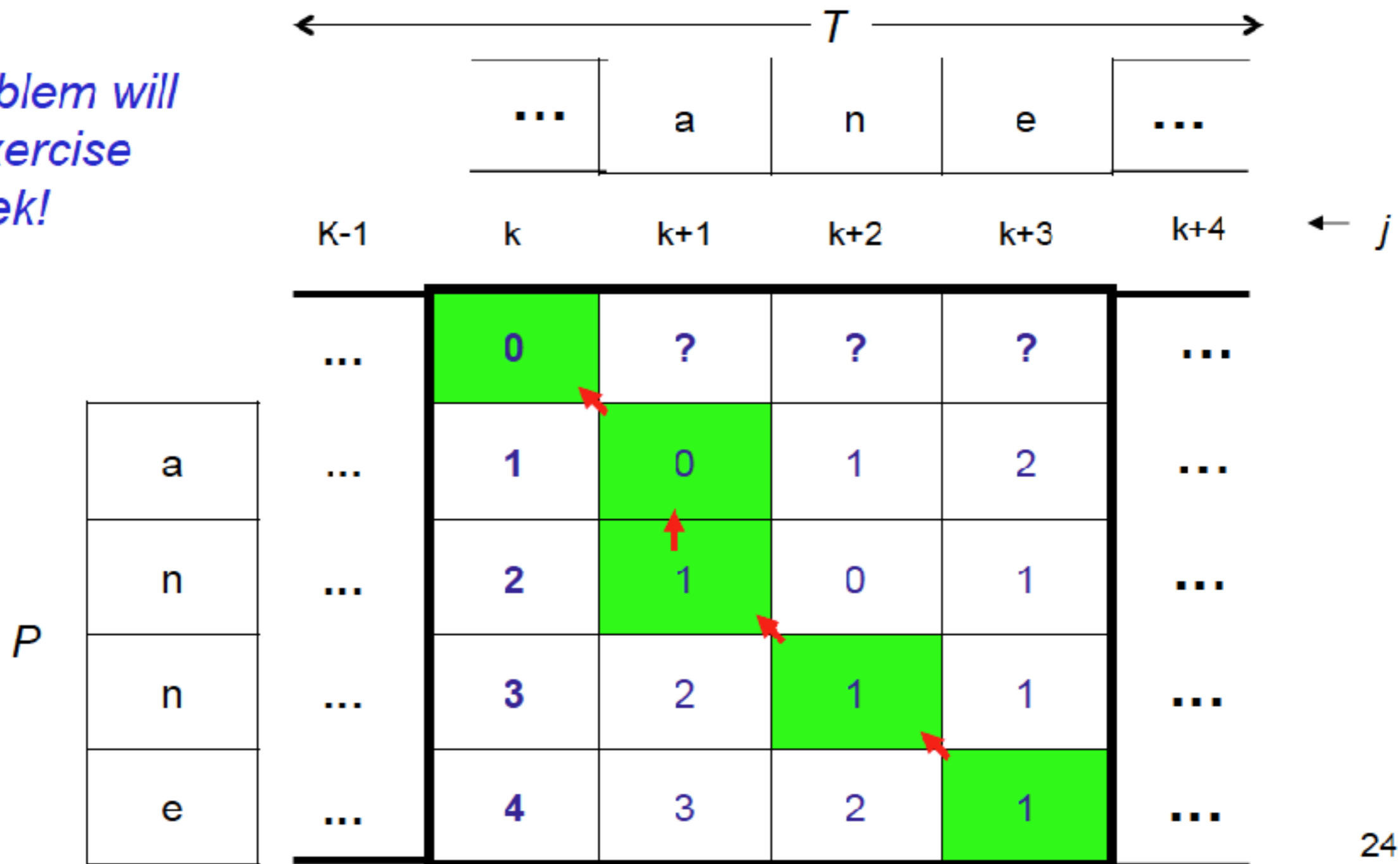
The result can be visualized as follows:

a	n	n	e
a	.	n	e

Until now we have computed the edit distance
between two strings P and T

But what about searching for substrings U of a long string T
so that $ED(P,U)$ is small, e.g, smaller than a given value?

*This problem will
be an exercise
next week!*



Relevance for research in genetics

Then T may be the full «genome» of one organism, and P a part of the genome of another.

Question: Does a sequence similar to P occur in T?

A chimpanzee gene (probably much longer):

u t t x v

The human genome (around 3×10^9 letters):

b s u t t v r t o f i g u t t v x | b s k u t t z x v k l h u u t t x v n x u t z t x v w

- Does the chimpanzee gene occur here, may be with a little change?
- Hopefully, *Torbjørn Rognes* from Bioinformatics will tell us more about such problems in a guest lecture (one hour) later this semester.

About Dynamic Programming in general

- Dynamic programming is typically used to solve optimization problems.
- The instances of the problem must be handled from smaller to larger ones, and the smallest (or simplest) instances can usually easily be solved directly (and be used for initialization of a program)
- For each problem instance I there is a set of instances I_1, I_2, \dots, I_k , all smaller than I , so that we can find an (optimal) solution to I if we know the (optimal) solution of all the problems I_1, I_2, \dots, I_k

The values of the yellow area are all computed when the gray value is to be computed. Usually only a few is used for computing each new entry

	0	1	...	$j-1$	j					
0	0	1		$j-1$	j					
1	1									
\vdots										
$i-1$	$i-1$									
i	i									

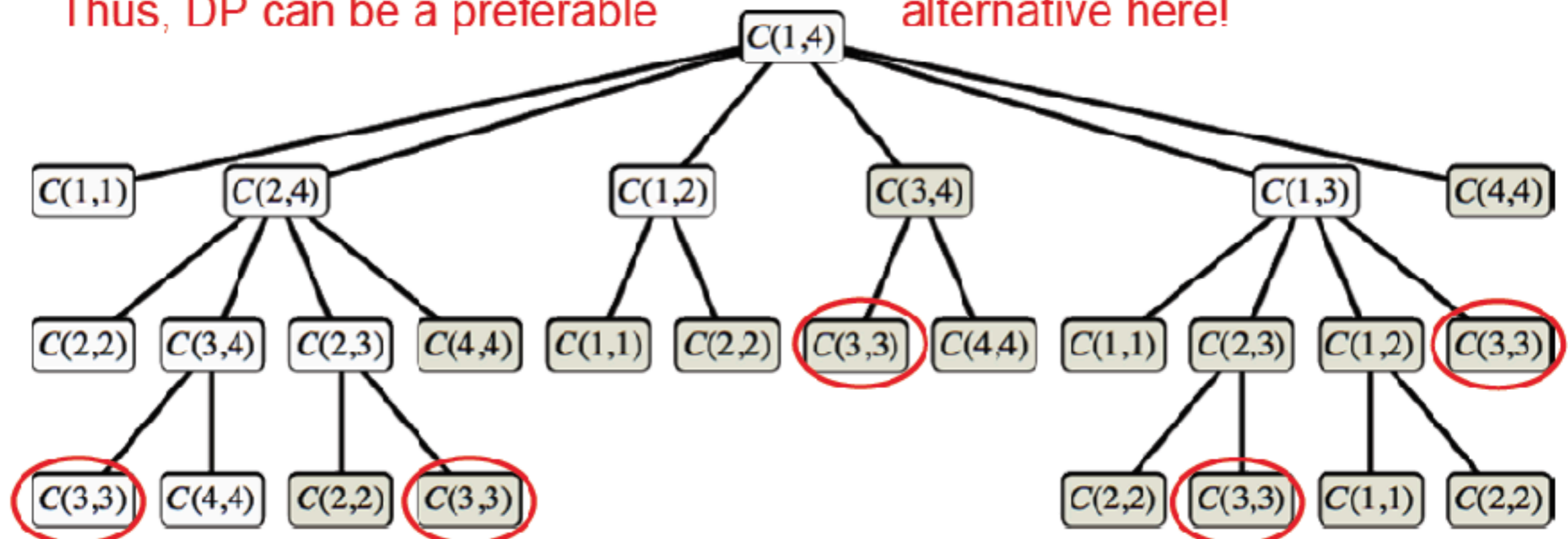
When should we use dynamic programming?

- Dynamic programming is useful if the total number of smaller instances needed to solve an instance I is so small that
 - The answer to all of them can be stored in a suitable table
 - They can be computed within reasonable time
- The main trick is to store the solutions in the table for later use. The real gain comes when each «smaller» table entry is used a number of times for later computations.

	0	1	...	$j-1$	j					
0	0	1		$j-1$	j					
1	1									
\vdots										
$i-1$	$i-1$									
i	i									

Another (slightly abstract) example

- As indicated on the previous slide, Dynamic Programming is more useful if the solution to a certain instance is used in the solution of many (larger) instances (assuming that the **size** of an instance $C(i, j)$ is $j - i$)
- In the problem C below, an instance is given by some data (e.g two strings) and by two integers i and j . Assume the corresponding instances are written $C(i, j)$. Thus the solutions to the instances can be stored in a two-dimensional table with dimensions i and j . (The **size** of an instance $C(i, j)$ is here $j - i$).
- Below, the children of a node N indicate the instances that we need the solution of, to compute the solution to instance N . We would therefore get this tree if we use recursion without remembering computed values at all.
- Note that the solution to many instances, e.g. $C(3,3)$, is used multiple times. **Thus, DP can be a preferable alternative here!**



A rather formal basis for Dynamic Programming

You don't need to learn this formalism and terminology

Assume we have a problem P with instances I_1, I_2, I_3, \dots

Dynamic programming might be useful for solving P , if:

- Each instance has a «size», where the «simplest» instances have small sizes, usually 0 or 1. (In our last example we can choose $m+n$ as the size)
- The (optimal) solution to instance I is written $s(I)$
- For each I there is a set of instances $\{J_1, \dots, J_k\}$ called the *base of I* , written $B(I) = \{J_1, J_2, \dots, J_k\}$ (where k may vary with I), and every J_k is smaller than I .
- We have a process/function *Combine* that takes as input an instance I , and the solutions $s(J_i)$ to all J_i in $B(I)$, so that

$$s(I) = \text{Combine}(I, s(J_1), s(J_2), \dots, s(J_k))$$

This is called the «recurrence relation» of the problem.

- For an instance I , we can set up a sequence of instances $\langle I_0, I_1, \dots, I_m \rangle$ with growing sizes, and where I_m is the problem we want to solve, and so that for all $p \leq m$, all instances in $B(I_p)$ occur in the sequence *before* I_p .
- The solutions of the instances I_0, I_1, \dots, I_m can be stored in a table of reasonable size compared to the size of the instance I .

Two variants of dynamic programming: Bottom up (traditional) and top down (memoization)

1. Traditional Dynamic Programming (bottom up)

- DP is traditionally performed bottom-up. All relevant smaller instances are solved first (independantly of whether they will be used later!), and their solutions are stored in the table.
- This usually leads to very simple and often rapid programs.

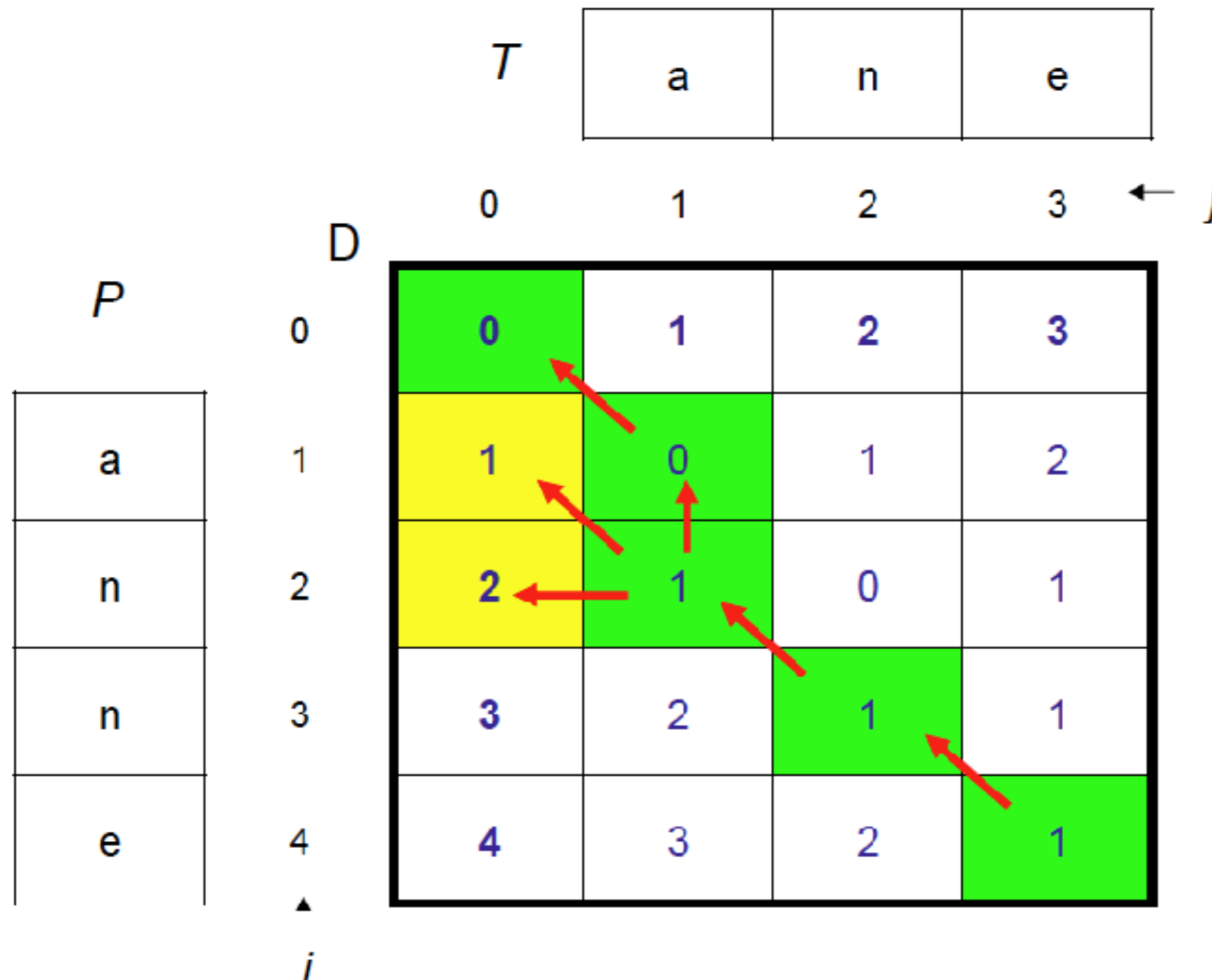
2. «Top-Down» Dynamic Programming

A drawback with traditional dynamic programming is that one usually solves a number of smaller instances that turn out not to be needed for the actual (larger) instance that we are really interested in.

- We can instead start at the (large) instance we want to solve, and do the computation recursively top-down. Also here we put computed solutions in the table as soon as they are computed.
- Each time we need to find the answer to an instance we first check in the table whether it is already solved, and if so we only use the stored solution. Otherwise we do recursive calls, and store the solution
- The table entries then need a special marker «not computed», which also should be the initial value of the entries.

«Top-Down» dynamic programming: "Memoization"

1. Start at the instance you want to solve, and ask recursively for the solution to the instances needed. The recursion will follow the red arrows in the figure below
2. As soon as you have an answer, fill it into the table, and take it from there when the answer to the same instance is later needed.



Benefit:

You only have to compute the needed table entries (those colored to the left)

But:

Managing the recursive calls take some extra time, so it does not always execute fastest.³¹

A type of problems typically solved by DP

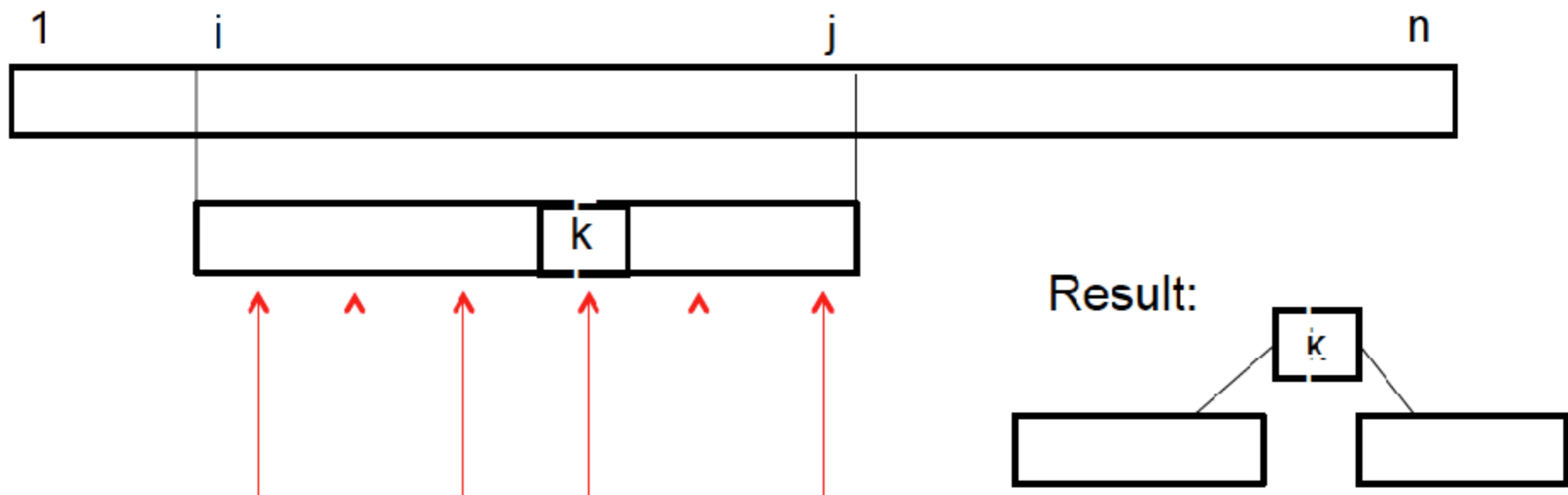
Both the «optimal matrix multiplication» and «optimal search tree» problem is of this type with only small modification!

- Assume we have a sequence of elements that should be turned into an optimal binary tree according to some criterium.
- Assume the sequence occur in an array $E[1:n]$, and are $\langle e_1, e_2, \dots, e_n \rangle$
- We assume:
 1. What is an optimal subtree containing the interval of elements $\langle e_i, \dots, e_j \rangle$, written $E[i, j]$, depends only on the values of e_i, \dots, e_j themselves (and maybe of some «static» global information or table)
 2. The optimal subtree of $E[i, j]$ will have one of the elements in $E[i, j]$ as root, say e_k , and have the optimal subtrees of the intervals $E[i, k-1]$ and $E[k+1, j]$ as subtrees
 3. The «quality» of an optimal subtree for $E[i, j]$ is written $Q(i, j)$.
 4. There is also a formula $Q'(i, k, j)$ that computes the quality of the tree formed by using the element e_k as root. This should only depend on $Q(i, k-1)$, $Q(k+1, j)$ and the value of e_k .
 5. That the quality of an empty tree and a tree with one element can be computed. This will make up the initialization of the malgorithm below.

Typical DP problem 2

We can then use DP to compute $Q(1,n)$ (and the optimal tree for $E[1, n]$) by computing the $Q(i, j)$ for the smallest intervals first, by the following recurrence formula:

$$Q(i, j) = \max \text{ over } k = i, i+1, \dots, j \text{ of } Q'(i, k, j)$$



Try each of these as root for this interval (try all $k = i \dots j$). Find the best k (that is, largest $Q'(i,k,j)$), and choose this as root. You will all the time know the answer for the shorter intervals ($[i, k-1]$ and $[k+1, j]$) during this computation.

Ch. 9.2. Optimal Matrix Multiplication Order

Given the sequence M_0, M_1, \dots, M_{n-1} of matrices. We want to compute the product: $M_0 \cdot M_1 \cdot \dots \cdot M_{n-1}$.

Note that, for this multiplication to be meaningful, the length of the rows in M_i must be equal to the length of the columns M_{i+1} for $i = 0, 1, \dots, n-2$

Matrix multiplication is *associative*: $(A \cdot B) \cdot C = A \cdot (B \cdot C)$

But it is *not symmetric*, since $A \cdot B$ generally is different from $B \cdot A$

Thus, one can do the multiplications in different orders. E.g., with four matrices it can be done in the following five ways (where only those corresponding to **binary** trees are allowed):

$$\begin{aligned} &(M_0 \cdot (M_1 \cdot (M_2 \cdot M_3))) \\ &(M_0 \cdot ((M_1 \cdot M_2) \cdot M_3)) \\ &((M_0 \cdot M_1) \cdot (M_2 \cdot M_3)) \\ &((M_0 \cdot (M_1 \cdot M_2)) \cdot M_3) \\ &(((M_0 \cdot M_1) \cdot M_2) \cdot M_3) \end{aligned}$$

The cost (the number of simple (scalar) multiplications) for these will usually vary a lot between the different alternatives. We want to find the one with as few scalar multiplications as possible.

Optimal matrix multiplication order, 2

Given two matrices A and B with dimensions:

A is a $p \times q$ matrix,

B is a $q \times r$ matrix.

The cost of computing $A \cdot B$ is $p \cdot q \cdot r$, and the result is a $p \times r$ matrix

Example showing that the multiplication order has significance:

Compute $A \cdot B \cdot C$, where

A is a 10×100 matrix, B is a 100×5 matrix, and C is a 5×50 matrix.

Computing $D = (A \cdot B)$ costs 5,000 and gives a 10×5 matrix.

Computing $D \cdot C$ costs 2,500.

Total cost for $(A \cdot B) \cdot C$ is thus 7,500.

Computing $E = (B \cdot C)$ costs 25,000 and gives a 100×50 matrix.

Computing $A \cdot E$ costs 50,000.

Total cost for $A \cdot (B \cdot C)$ is thus 75,000.

We would indeed prefer to do it the first way!

Optimal matrix multiplication order, 3

Given a sequence of matrices M_0, M_1, \dots, M_{n-1} . We want to find the cheapest way to do this multiplication (that is, an «optimal paranthesization»).

From the outermost level, the first step in a parenthesization is a partition into two parts: $(M_0 \cdot M_1 \cdot \dots \cdot M_k) \cdot (M_{k+1} \cdot M_{k+2} \cdot \dots \cdot M_{n-1})$

If we know the best parenthesization of the two parts, **we can sum their cost and add the pqr-cost for the last multiplication**, and thereby get the smallest cost, given that we have to use this outermost partititon.

Thus, to find the best outermost parenthesization of M_0, M_1, \dots, M_{n-1} , we can simply look at all the $n-1$ possible outermost partitions ($k = 0, 1, n-2$), and choose the best. But we will then need the cost of the optimal parenthesization of a lot of instances of smaller sizes.

And we shall say that the **size** of the instance M_i, M_{i+1}, \dots, M_j is $j - i$.

We therefore generally have to look at the best parenthesization of all intervals M_i, M_{i+1}, \dots, M_j , *in the order of growing sizes*.

We will refer to the lowest possible cost for the multiplication $M_i \cdot M_{i+1} \cdot \dots \cdot M_j$ as $m_{i,j}$

Optimal matrix multiplication order, 4

Let d_0, d_1, \dots, d_n be the dimensions of the matrices M_0, M_1, \dots, M_{n-1} , so that matrix M_i has dimension $d_i \times d_{i+1}$

As on the previous slide:

Let m_{ij} be the cost of an optimal parenthesization of M_i, M_{i+1}, \dots, M_j .

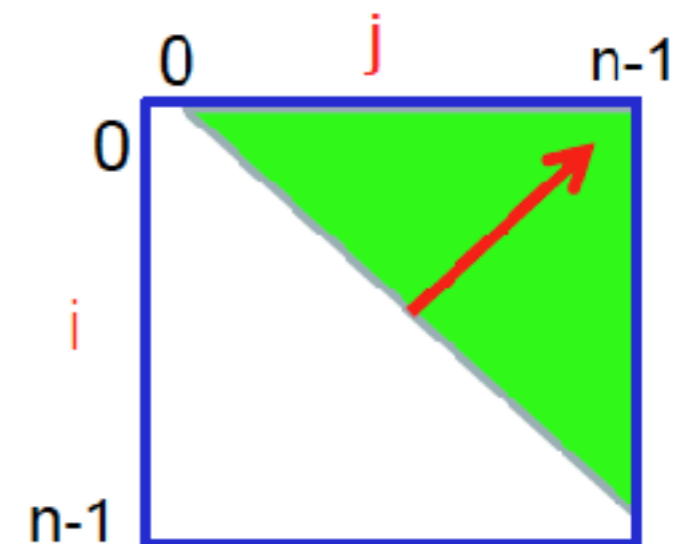
Thus the value we are interested in is $m_{0,n-1}$

The recurrence relation for m_{ij} will be:

$$m_{i,j} = \min_{i \leq k < j} \{ m_{i,k} + m_{k+1,j} + d_i d_{k+1} d_{j+1} \} \quad \text{when } 0 \leq i < j \leq n-1$$

$$m_{i,i} = 0 \quad \text{when } 0 \leq i \leq n-1$$

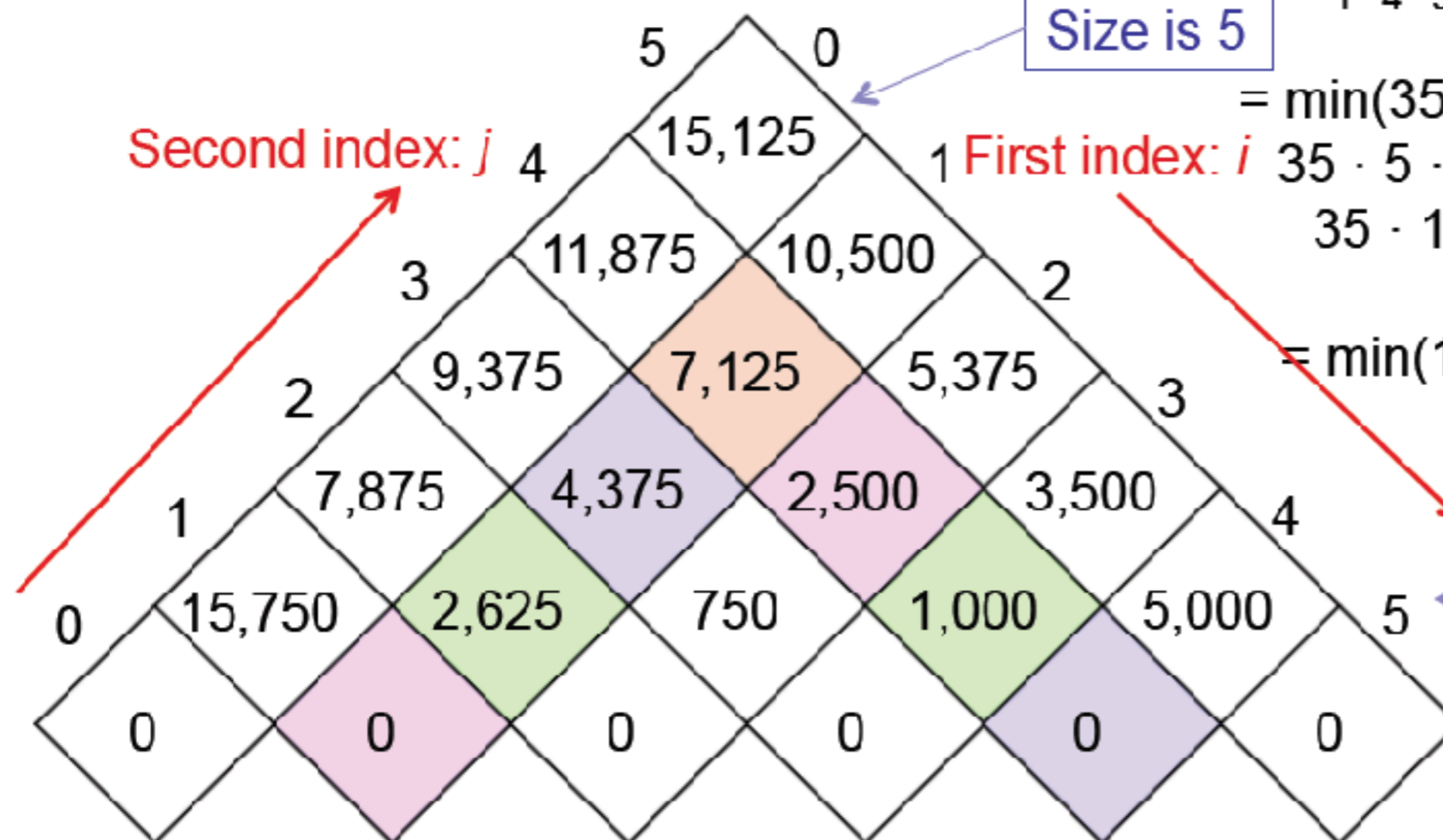
Here, importantly, the values $m_{k,l}$ that we need for computing m_{ij} are all for *smaller* instances. With usual indexing this means we shall fill the green area from the diagonal towards the upper right corner, as shown by the red arrow. In the next slide this green triangle is turned 45 degrees against the clock.



The table: Optimal matrix multiplication order

d	30	35	15	5	10	20	25
-----	----	----	----	---	----	----	----

The values $m_{i,j}$:



Example:

$$m_{1,4} = \min(d_1 d_2 d_5 + m(1,1) + m(2,4), \\ d_1 d_3 d_5 + m(1,2) + m(3,4), \\ d_1 d_4 d_5 + m(1,3) + m(4,4))$$

$$= \min(35 \cdot 15 \cdot 20 + 0 + 2,500, \\ 35 \cdot 5 \cdot 20 + 2,625 + 1,000, \\ 35 \cdot 10 \cdot 20 + 4,375 + 0)$$

$$= \min(13000, 7125, 11375)$$

$$= 7125$$

Size is 1

Size is 0

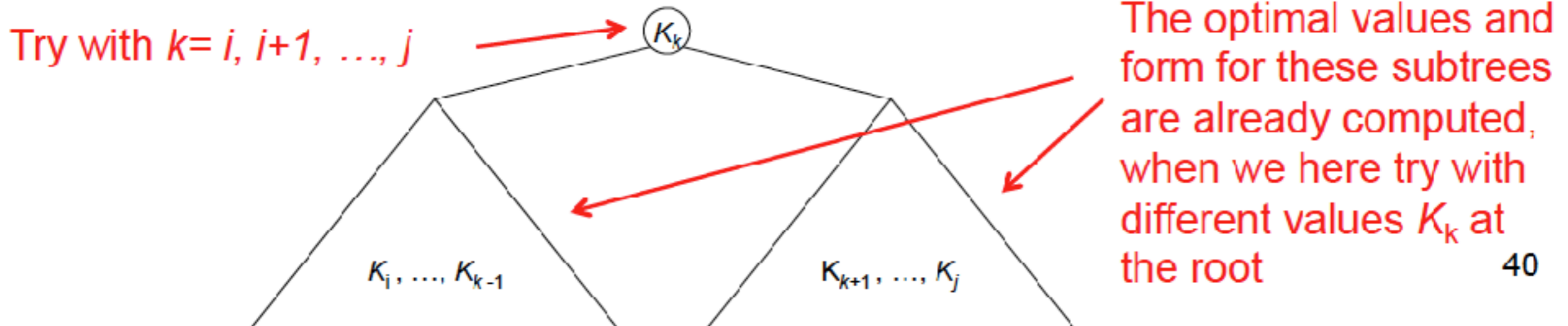
Definition: Size of the instance covering the interval from pos. i to pos. j is $j - i$

Program: Optimal matrix multiplication order

```
function OptimalParenth( d[0 : n - 1] )  
  for i  $\leftarrow$  0 to n-1 do  
    m[i, i]  $\leftarrow$  0  
  for diag  $\leftarrow$  1 to n - 1 do  
    for i  $\leftarrow$  0 to n - 1 - diag do  
      j  $\leftarrow$  i + diag  
      m[i, j]  $\leftarrow$   $\infty$  // Relative to the scalar values that can occur  
      for k  $\leftarrow$  i to j - 1 do  
        q  $\leftarrow$  m[i, k] + m[k + 1, j] + d[i]  $\cdot$  d[k + 1]  $\cdot$  d[j + 1]  
        if q < m[i, j] then  
          m[i, j]  $\leftarrow$  q  
          c[i, j]  $\leftarrow$  k  
        endif  
      endfor  
    endfor  
  endfor  
  return m[0, n - 1]  
end OptimalParenth
```

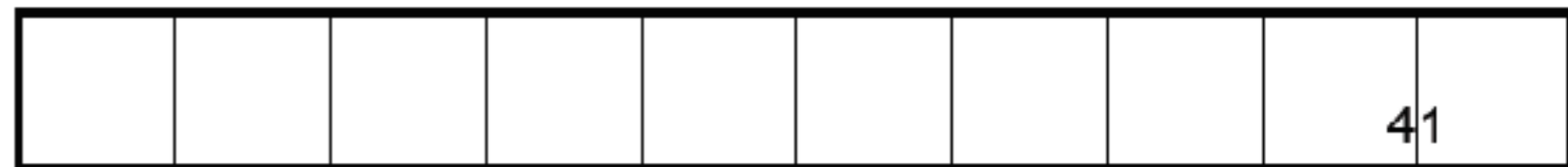
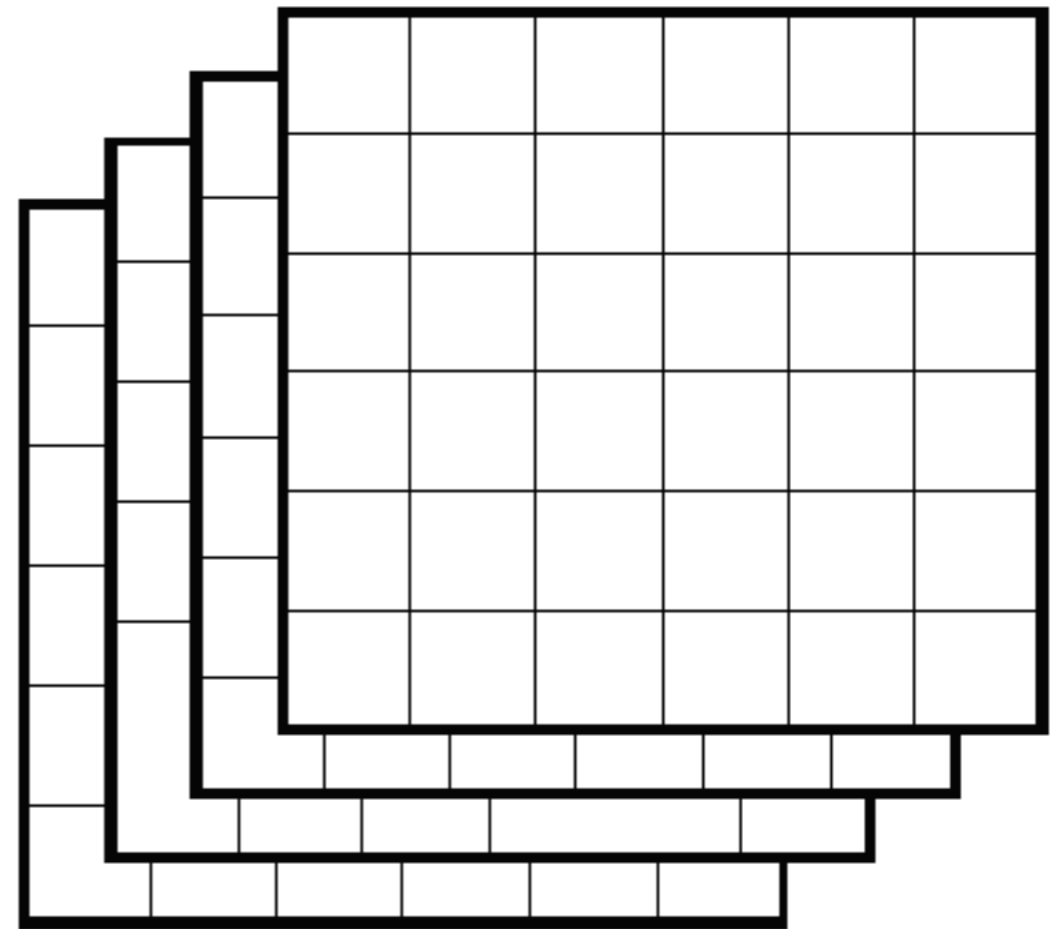
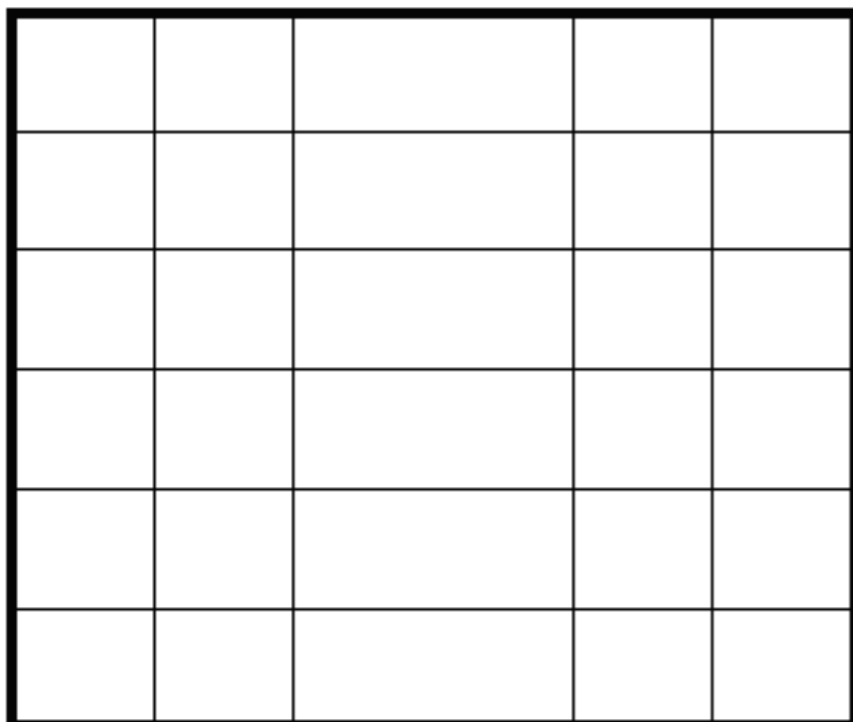
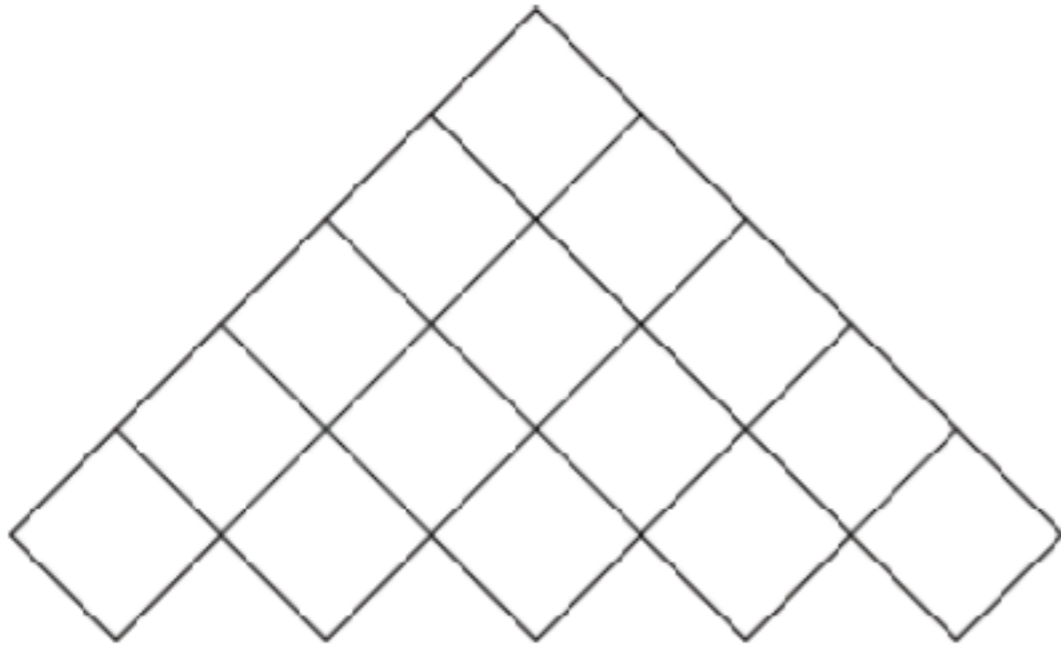
Ch. 9.3. Optimal search trees

- To get a manageable problem that still catches the essence of the general problem, we shall assume that all q-es are zero (that is, we never search for values not in the tree)
- A key to a solution is that a subtree in a search tree will always represent an interval of the values in the tree in sorted order (and that such an interval can be seen as an optimal search instance in itself)
- Thus, we can use the same type of table as in the matrix multiplication case, where the value of the optimal tree over the values from index i to index j is stored in $A[i, j]$, and the size of such an instance is $j - i$
- Then, for finding the optimal tree for an interval with values K_i, \dots, K_j we can simply try with each of the values K_i, \dots, K_j as root, and use the best subtrees in each of these cases (whose optimal values are already found).
- To compute the cost of the subtrees is slightly more complicated than in the matrix case, but is no problem.



Dynamic programming in general:

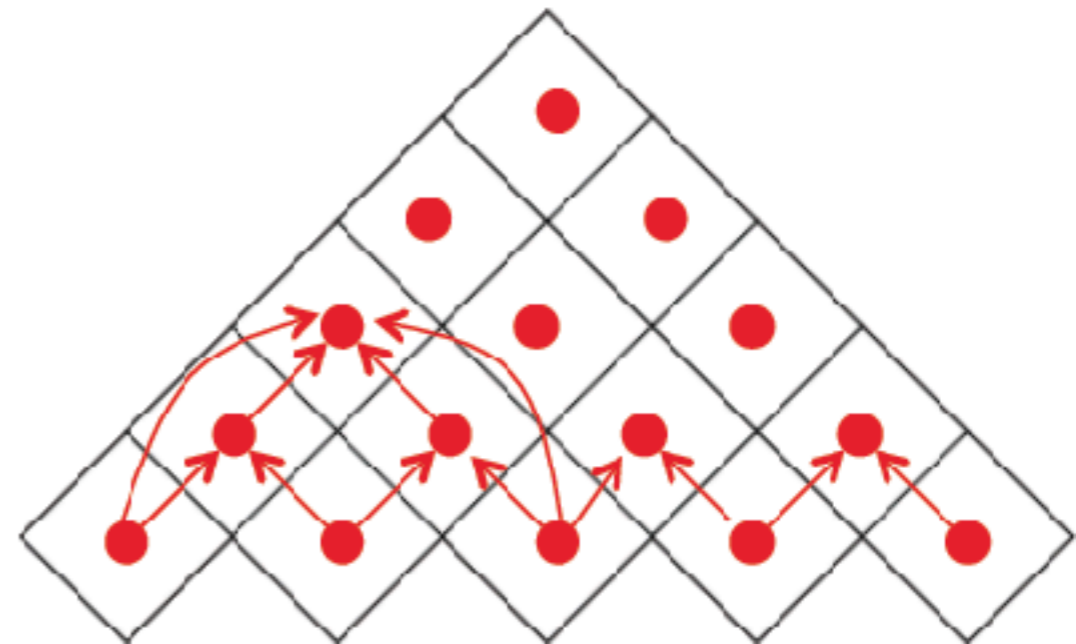
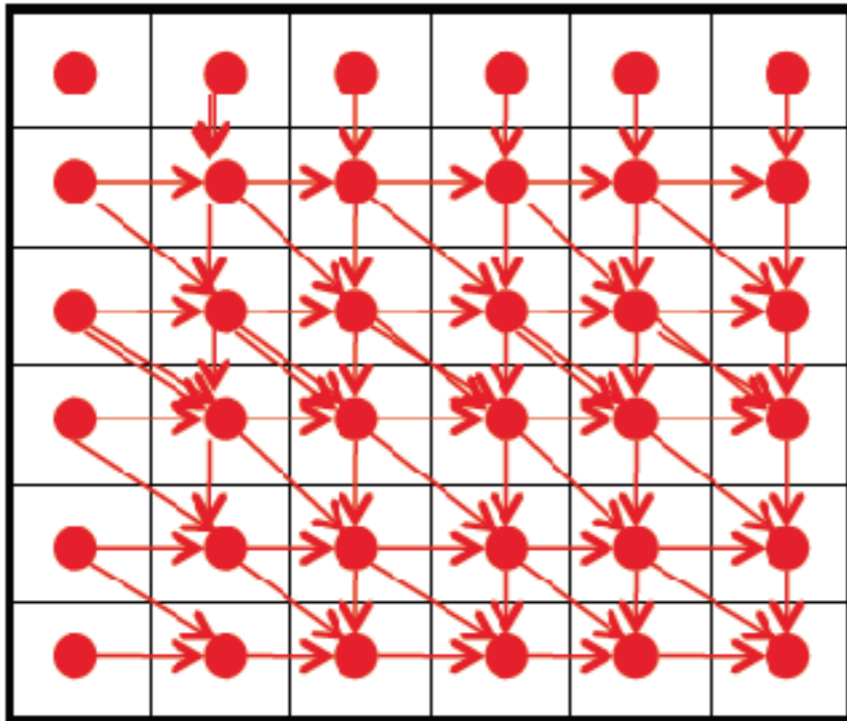
We fill in different types of tables «bottom up»
(smallest instances first)



Dynamic programming

Filling in the tables

- It is always safe to solve all the smaller instances before any larger ones, using the defined size of the instances.
- However, if we know what smaller instances are needed to solve a larger instance, we can deviate from the above. The important thing is that the smaller instances needed to solve a certain instance J is computed before we solve J .
- Thus, if we know the «dependency graph» of the problem (which must be cycle-free, see examples below), the important thing is to look at the instances in an order that conforms with this dependency. This freedom is often utilized to get a simple computation.



Thanks