

# INF4130: Dynamic Programming

September 6, 2016

- In the textbook: Ch. 9, and Section 20.5
  - We will use an example from Section 20.5 (Approximate string matching)
  - And we will study that example *before* we look at the theory
- The slides presented here have a different introduction to the theory than the textbook
  - This is done because the introduction in the textbook (built on its «principle of optimality») seems rather confusing.
  - Thus, some explanations for why the algorithms work will also be a little different than in the textbook.
- And the curriculum is the version used on these slides, not the introduction in the textbook.

# 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.



# Example: Given a string $T$ and a shorter string $P$ . Find strings «similar» to $P$ in $T$ (from Ch. 20.5)

We will first look at a similar problem, and leave the final step as an exercise  
This problem is similar to the one in 9.4: «Longest Common Subsequence»

A string  $P$  is a  $k$ -approximation of a string  $T$  if  $T$  can be converted to  $P$  by a sequence of maximum  $k$  of the following operations:

<b>Substitution</b>	One symbol in $T$ is changed to another symbol.
<b>Addition</b>	A new symbol is inserted somewhere in $T$ .
<b>Removing</b>	One symbol is removed from $T$ .

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

The smallest number of such operations needed to convert  $T$  to  $P$

*(or  $P$  to  $T$ ! Note that the definition is symmetric in  $P$  and  $T$ !)*

**Example.**

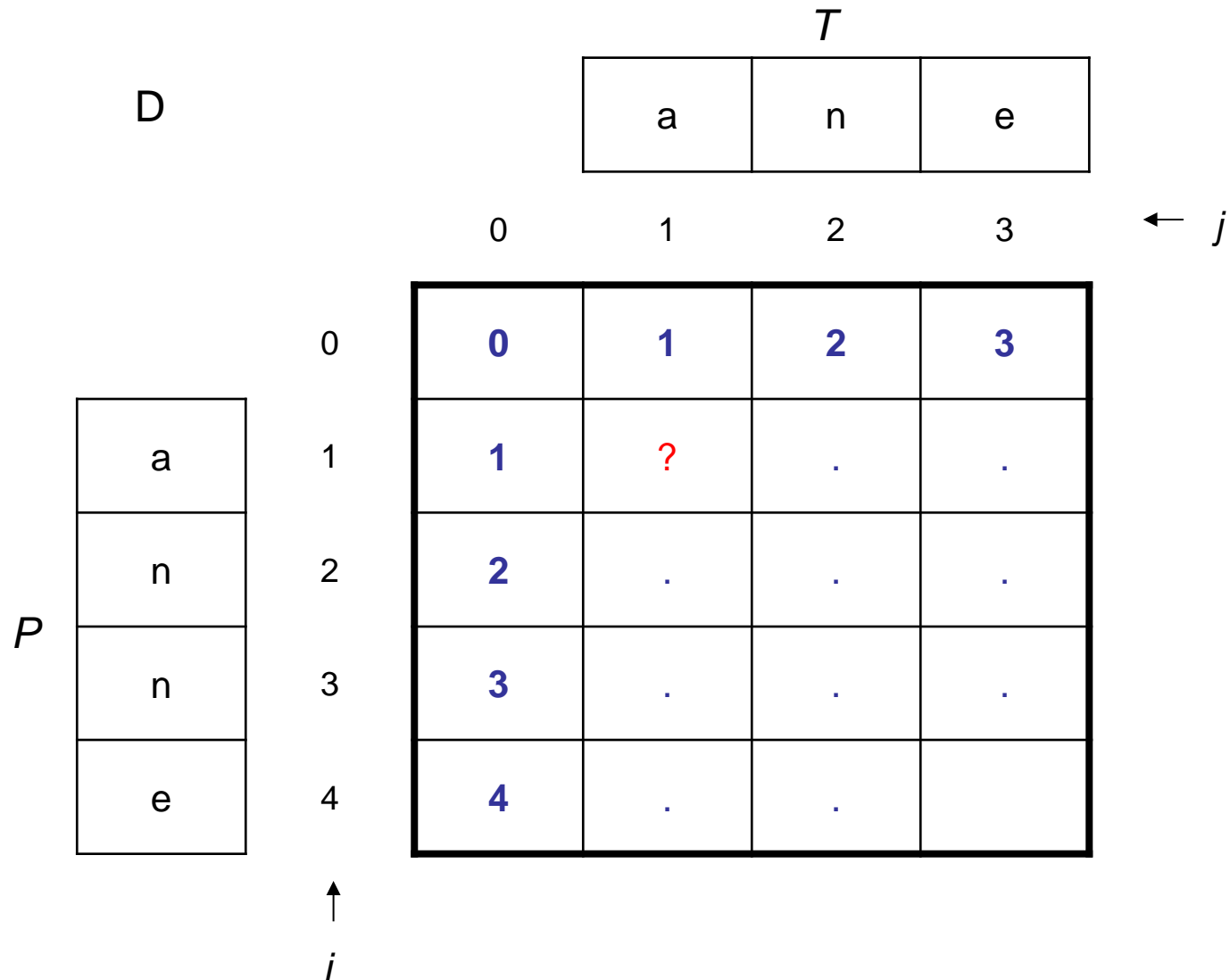
logarithm  $\rightarrow$  alogarithm  $\rightarrow$  algorithm  $\rightarrow$  algorithm (Steps: +a, -o, a $\rightarrow$ o)  
 $T$   $P$

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



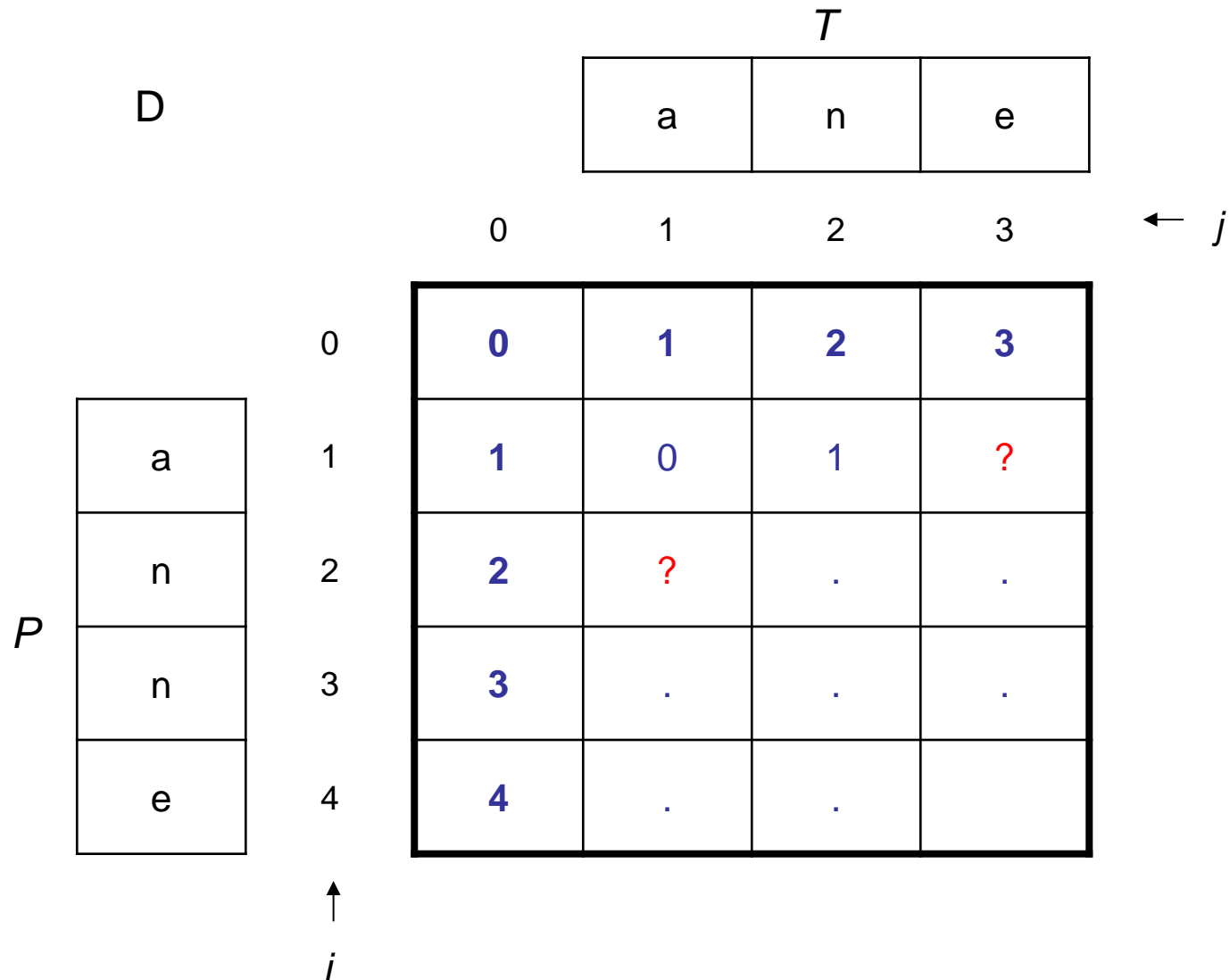
# Example: $P = \text{«anne»}$ and $T = \text{«ane»}$

- Some obvious values for  $D[i, j]$  are given below
- How to find more values?
- The answer  $ED(P, T)$  will appear in  $D[4, 3]$  (lower right)



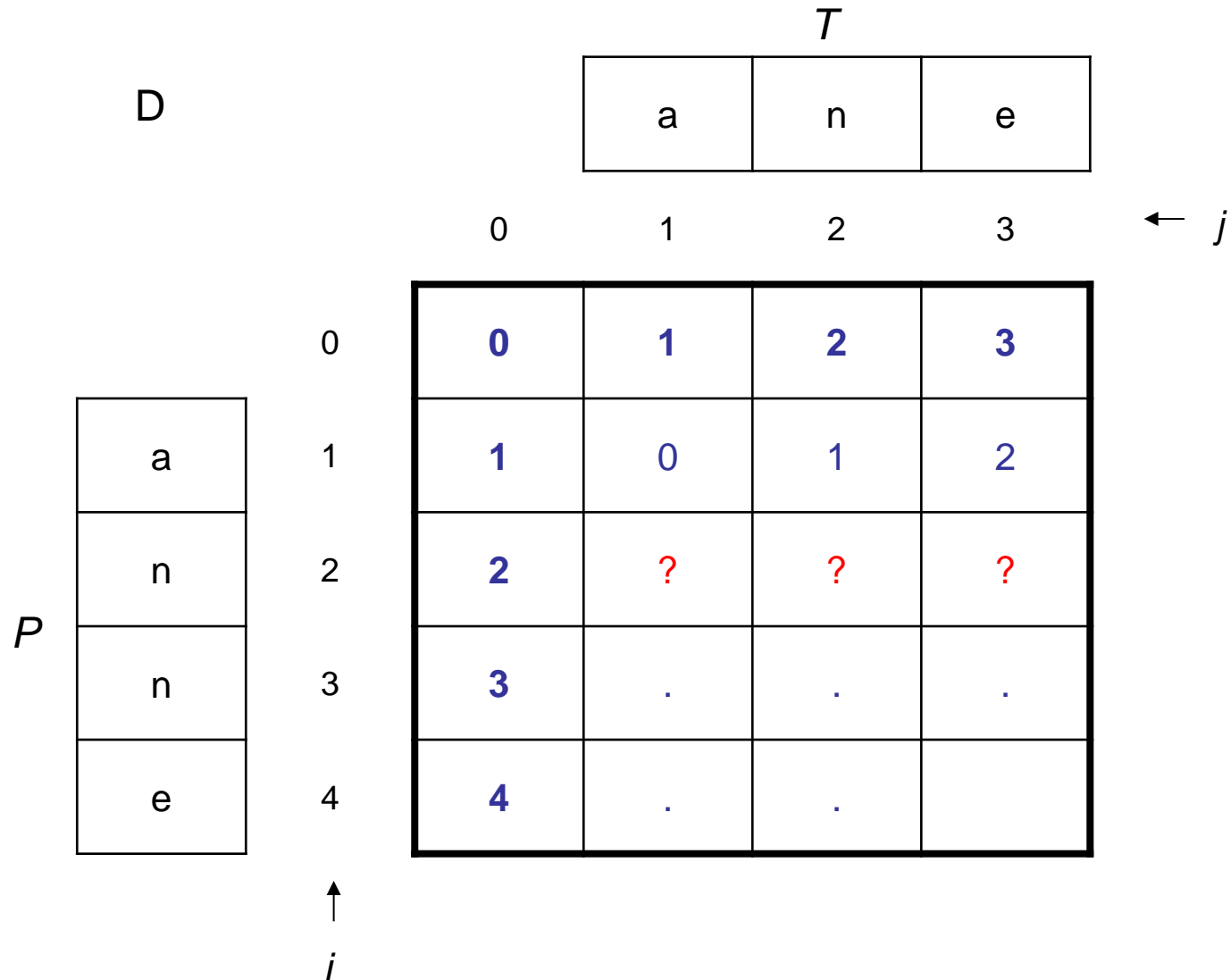
# Example: $P = \text{«anne»}$ and $T = \text{«ane»}$

- Some obvious values for  $D[i, j]$  are given
- How to find more values?
- The answer  $ED(P, T)$  will appear in  $D[4, 3]$  (lower right)



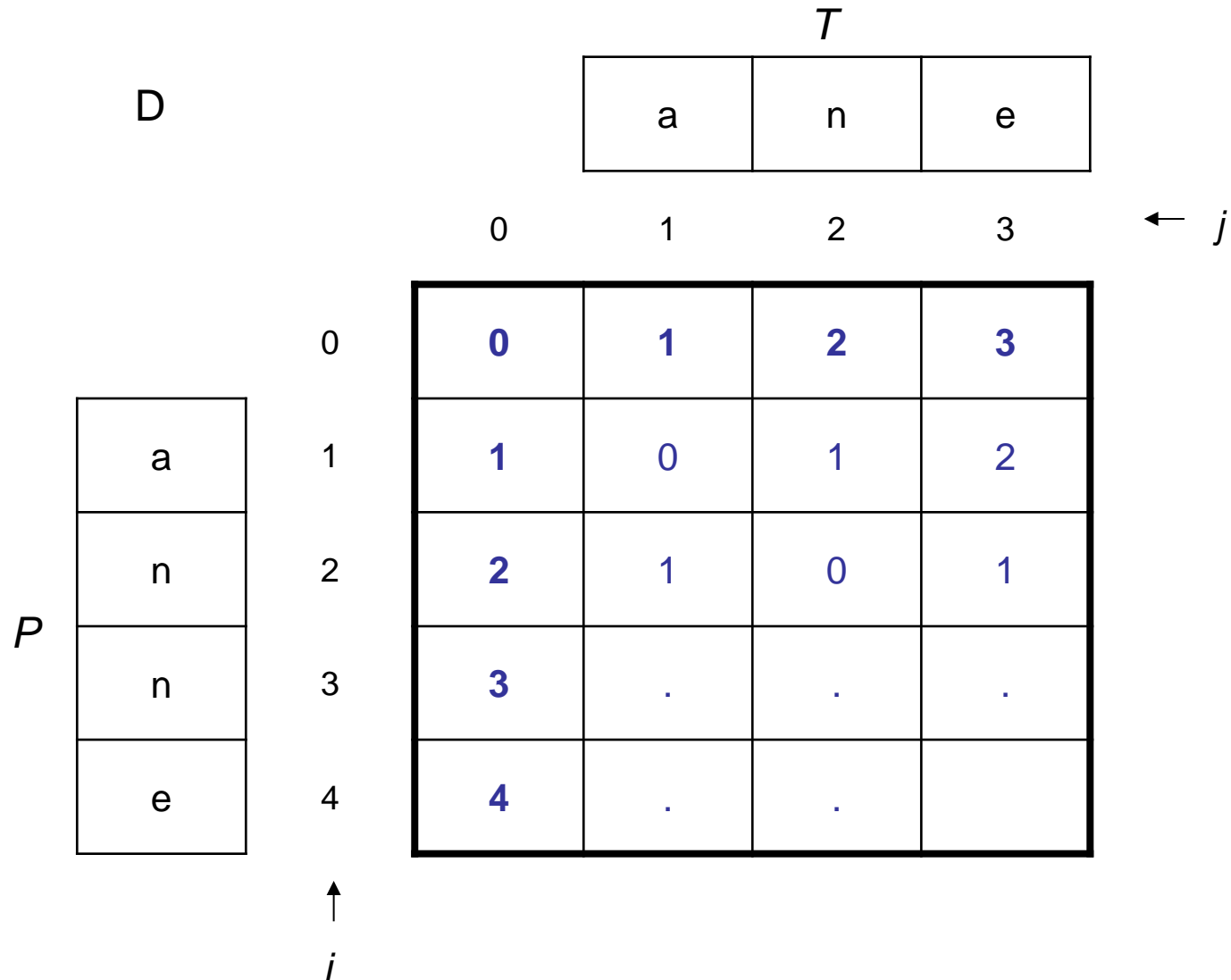
# Example: $P = \text{«anne»}$ and $T = \text{«ane»}$

- Some obvious values for  $D[i, j]$  are given
- How to find more values?
- The answer  $ED(P, T)$  will appear in  $D[4, 3]$  (lower right)



# Example: P = «anne» and T = «ane»

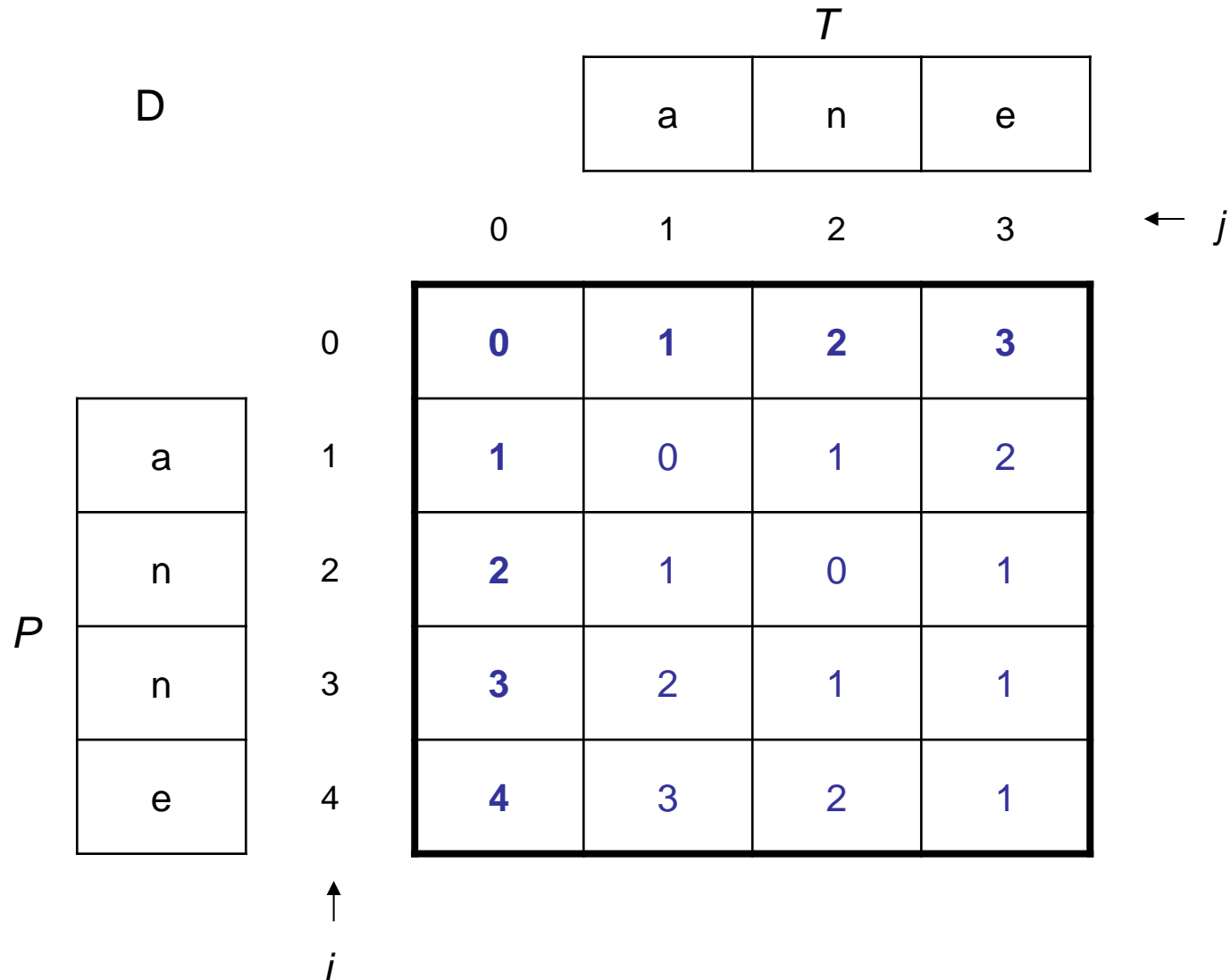
- Some obvious values for  $D[i, j]$  are given
- How to find more values?
- The answer  $ED(P, T)$  will appear in  $D[4, 3]$  (lower right)





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

- Some obvious values for  $D[i, j]$  are given
- How to find more values?
- The answer  $ED(P, T)$  will appear in  $D[4, 3]$  (lower right)



# The «recurrence relation»

To be able to fill in this matrix we shall use that we have the relation indicated below between neighbouring entries in  $D$ . **For proof see next slide.**

Note that the value of  $D[i,j]$  only depends on entries in  $D$  with «smaller» index-pairs:  $D[i-1,j-1]$ ,  $D[i-1,j]$ , and  $D[i,j-1]$

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

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

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


The matrix  $D$ :

		T <span style="color:red">→</span>												
		0	1	...	j-1	j					n			
P	0	0	1			j-1	j							n
	1	1												
	⋮													
	i-1	i-1												
	i	i					?							
	m	m												

# Proof of the recurrence relation

As indicated: The previous slides do not give a full proof for the equality in the recurrence relation (but only a «less than or equal»):

$$D[i, j] = \begin{cases} D[i-1, j-1] & \text{if } P[i] = T[j] \\ \min\{ \underbrace{D[i-1, j-1]+1}_{\text{substitution}}, \underbrace{D[i-1, j]+1}_{\text{addition to T}}, \underbrace{D[i, j-1]+1}_{\text{Deletion from T}} \} & \text{otherwise} \end{cases}$$



- This is because we can easily give  $D[i, j]$  steps to change P to T
- To prove the full equality will be an exercise this week.
- As a base for this proof we will assume that a minimal sequence of edit operations can always be written in the following form:

```

l o g a r i t h m
  ↓
a l   g o r i t h m
+   -   *
    
```

- Here ‘+’ = insertion, ‘-’ = deletion, and ‘\*’ = substitution.

# Figure for computing the edit distance (see program next slide)

We have the following expression for  $D[i, j]$ :

$$D[i, j] = \begin{cases} D[i-1, j-1] & \text{if } P[i] = T[j] \\ \min\{ \underbrace{D[i-1, j-1]+1}_{\text{substitution}}, \underbrace{D[i-1, j]+1}_{\text{addition to T}}, \underbrace{D[i, j-1]+1}_{\text{Deletion from T}} \} & \text{otherwise} \end{cases}$$

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

	0	1	...	j-1	j					n
0	0	1		j-1	j					
1	1									
⋮										
i-1	i-1									
i	i									
m										

When fully filled in, we will find the edit distance between T and P in  $D[m, n]$

# 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
  for  $j \leftarrow 1$  to  $n$  do  $D[0, j] \leftarrow j$    endfor

  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 at previous slide):

$(1,1)$   $(1,2)$  ...  $(1,n)$

$(2,1)$   $(2,2)$  ...  $(2,n)$

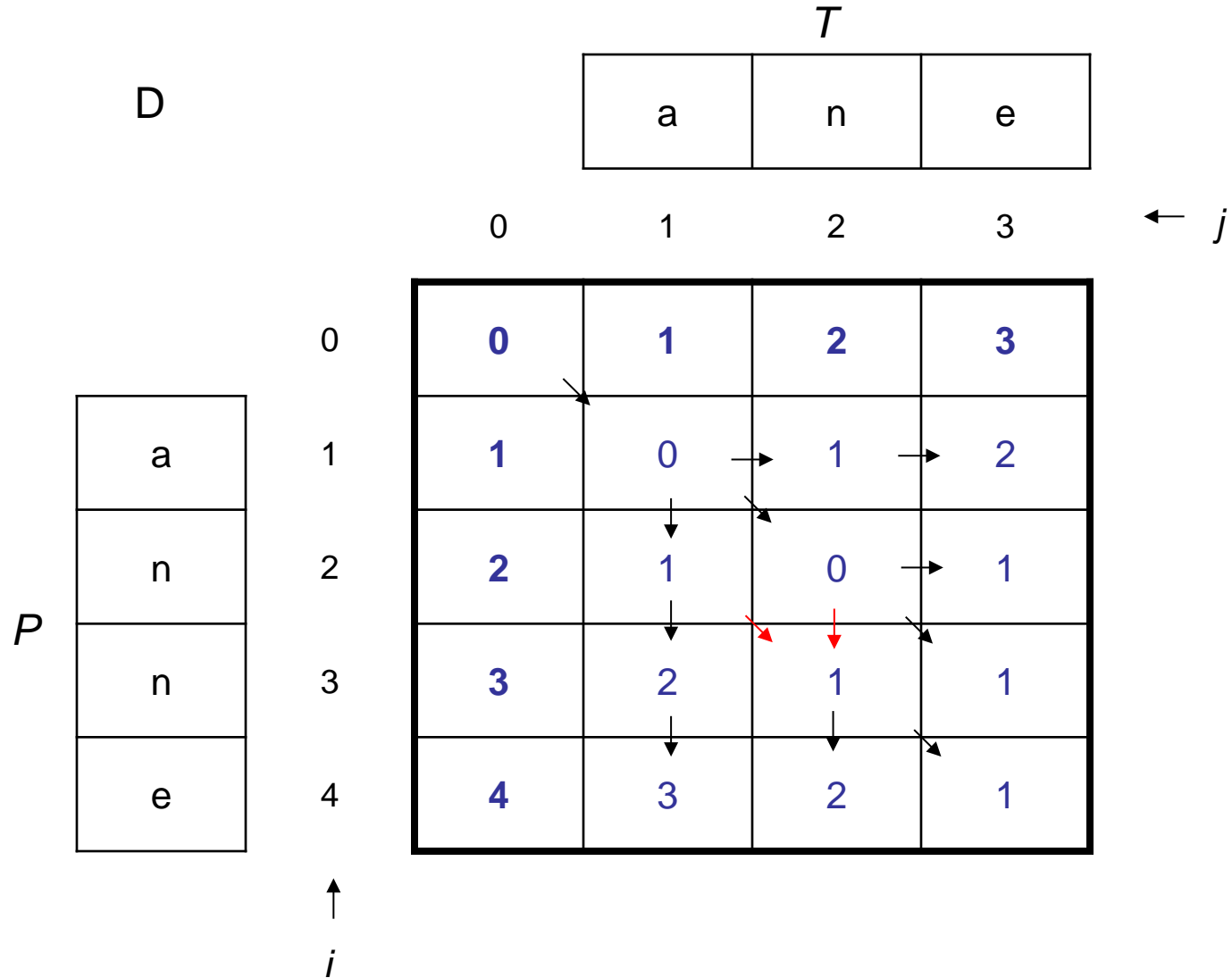
...

$(m,1)$  ...  $(m,n)$

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 computed before  $D[i, j]$ ,

# Our old example: Computing Edit distance



# Finding the edit steps

## Method:

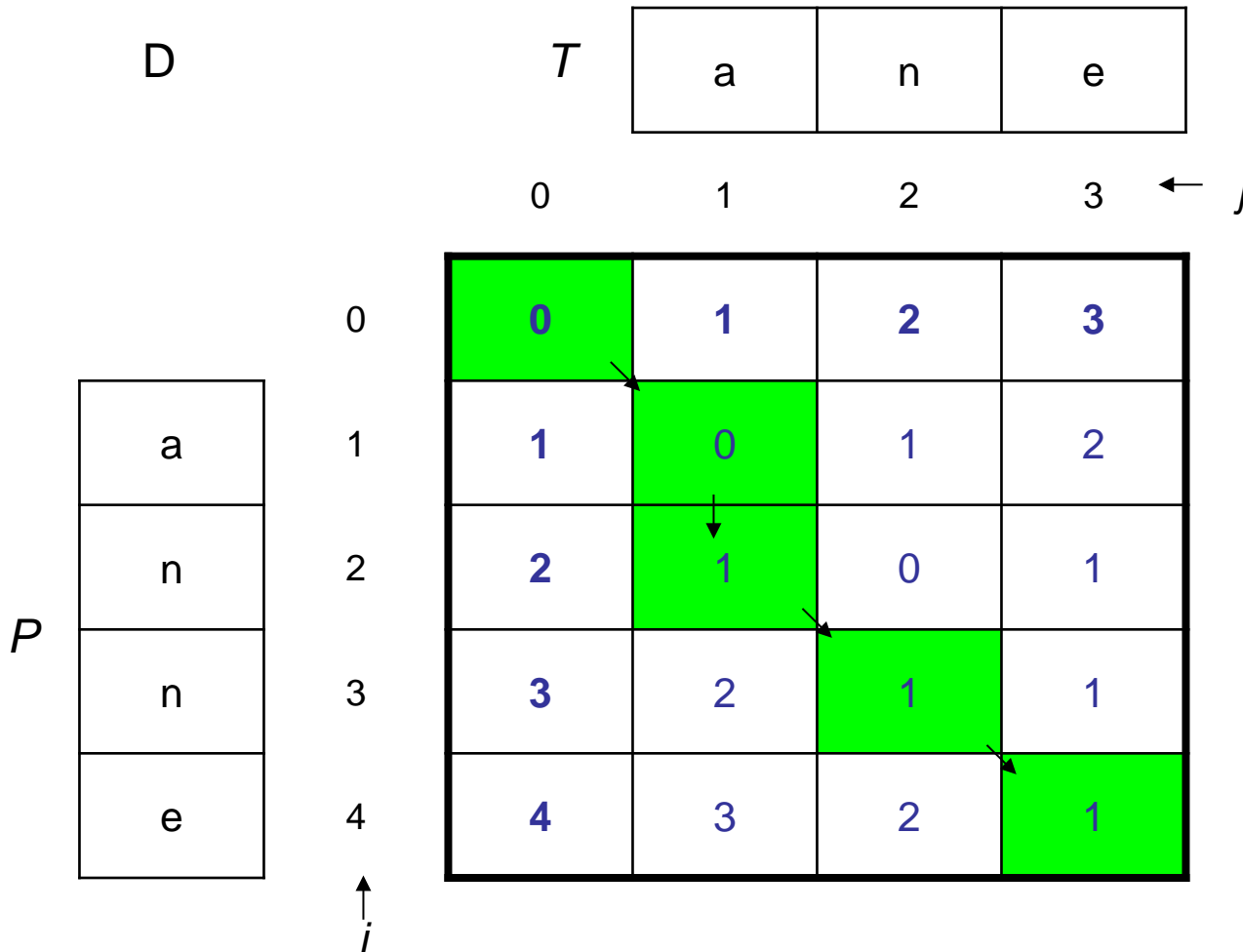
Follow the «path»  
used from the final  
entry back to [0,0]:

Diagonally, and  
 $P[i] = T[j]$ :  
- No edit needed

Diagonally, and  
 $P[i] \neq T[j]$ :  
- Substitution  
(Appear for  
 $D[3, 3]$ )

Downwards:  
- A letter is deleted  
from P

Towards the right:  
- A letter is added  
to P (Appear for  
 $D[1, 2]$ )



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

But what about searching for substrings  $U$  in  $T$  so that  $ED(P,U)$  is small, e.g, smaller than a certain given value?  
This will be solved as an exercise this week!

?

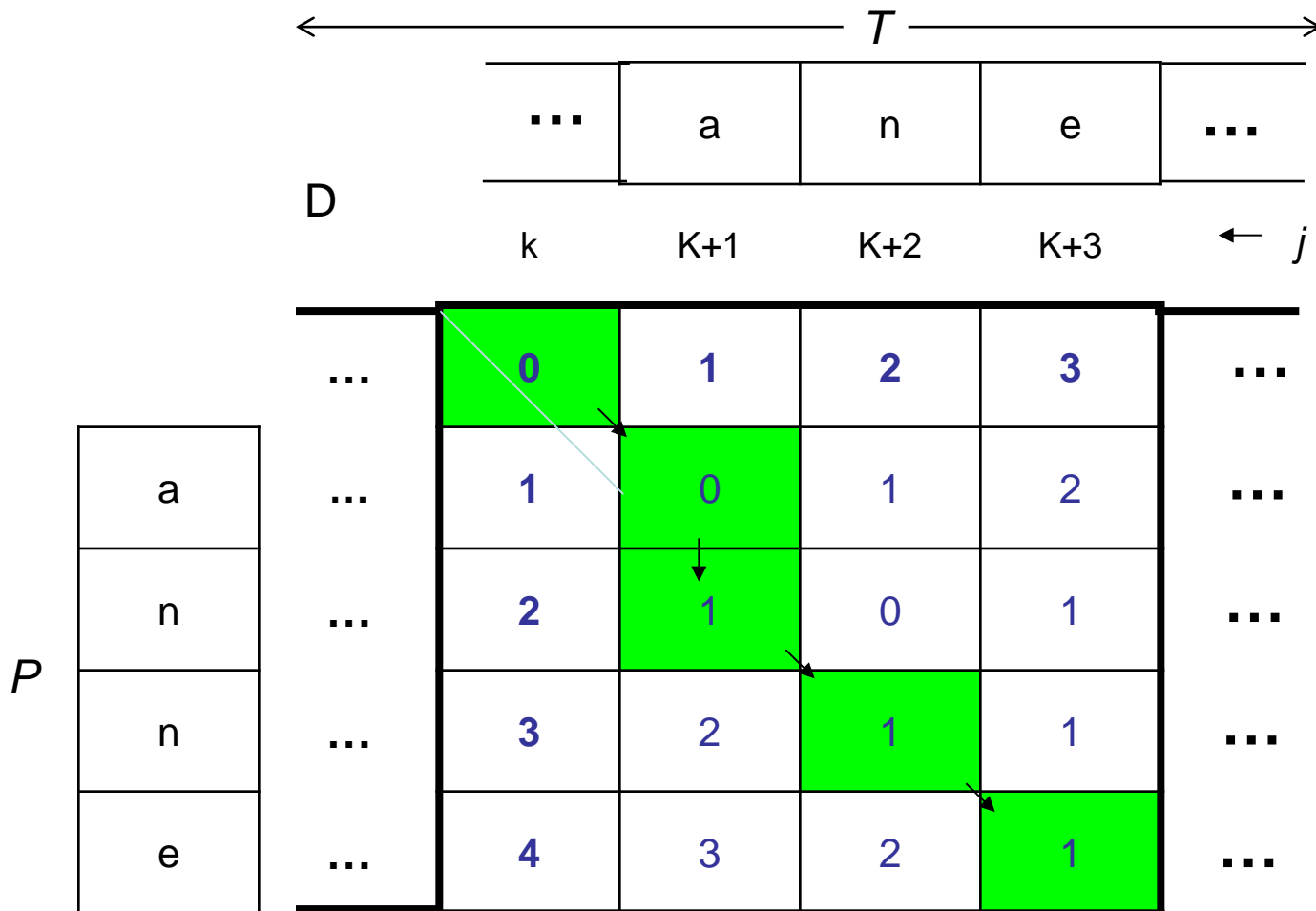




# Relevance for research in genetics

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 general formulation of Dynamic Programming

## Assume we have:

- A problem  $P$  with (usually infinitely) many instances  $I_1, I_2, I_3, \dots$
- An integer «size» for each instance, where the «simplest» instances have a small size, usually 0 or 1. (In our example the size can be chosen as  $m+n$ )
- The 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  $s(J)$  to all  $J$  in  $B(I)$ ,

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

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

- For an instance  $I$ , we can set up a sequence of instances  $\langle L_0, L_1, \dots, L_m = I \rangle$  so that for all  $p \leq m$ , all instances in  $B(L_p)$  occur in the sequence *before*  $L_p$ .
- The sequence can be stored in a suitable table of reasonable size compared to the size of the instance  $I$ .

# When to use dynamic programming?

- Dynamic programming is useful if the total number of smaller instances needed to solve an instance  $l$  (found recursively by the B-function) is so small that the answer to all of them can be stored in a suitable table.
- The main trick is to store the solutions in the table for later use. The real gain comes when each «smaller» entry is used a number of times.

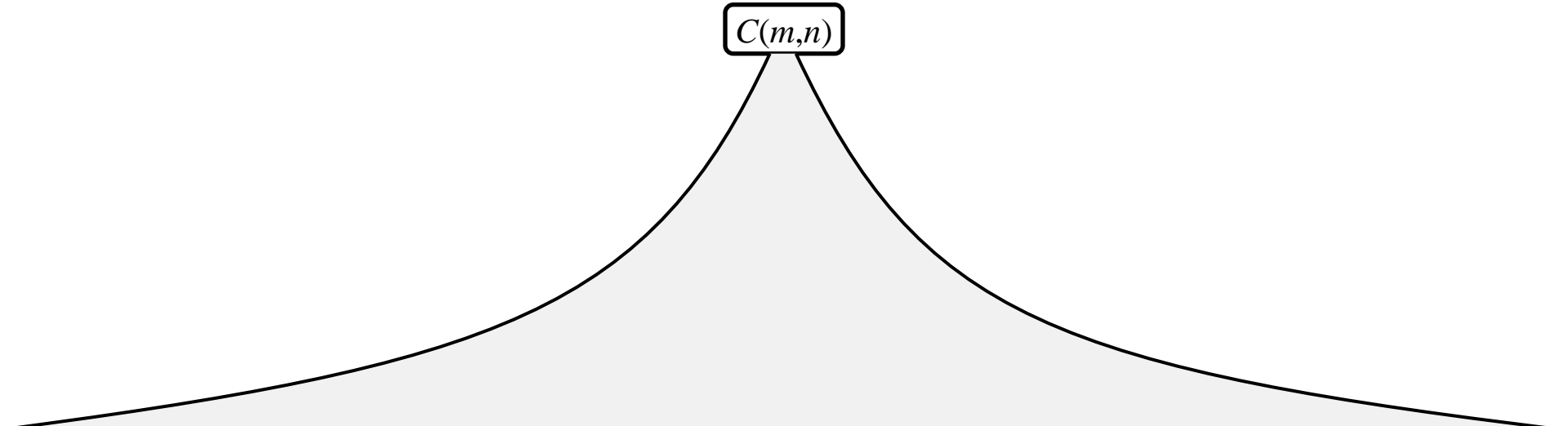
	0	1	...	$j-1$	$j$					
0	0	1		$j-1$	$j$					
1	1									
⋮										
$i-1$	$i-1$									
$i$	$i$									

The diagram shows a grid representing a dynamic programming table. The columns are labeled 0, 1, ...,  $j-1$ ,  $j$ . The rows are labeled 0, 1, ⋮,  $i-1$ ,  $i$ . The cell at row  $i-1$ , column  $j-1$  is shaded gray. Three arrows originate from this cell: one points to the cell at row  $i-1$ , column  $j$ ; one points to the cell at row  $i$ , column  $j-1$ ; and one points to the cell at row  $i$ , column  $j$ . The cell at row  $i$ , column  $j$  is highlighted in yellow.



# Number of smaller solutions needed

- If the number of solutions to smaller instances needed to find the solution to a certain instance is very big (e.g. exponential in the size of the instance), then the resulting DP algorithm will usually not be practical, as the table must be very big, and there will probably be little reuse of smaller solutions
- For Dynamic Programming to work well, this number should at most be polynomial in the size of the instance, and usually rather small.



$C(m,n)$

Sketch indicating a situation when the solution of one instance needs the solution of a large number of *different* smaller instances.

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

## Traditional Dynamic Programming (bottom up)

- Is traditionally performed bottom-up. All relevant smaller instances are solved first, and their solutions are stored in the table.
- This usually leads to very simple and often rapid programs.

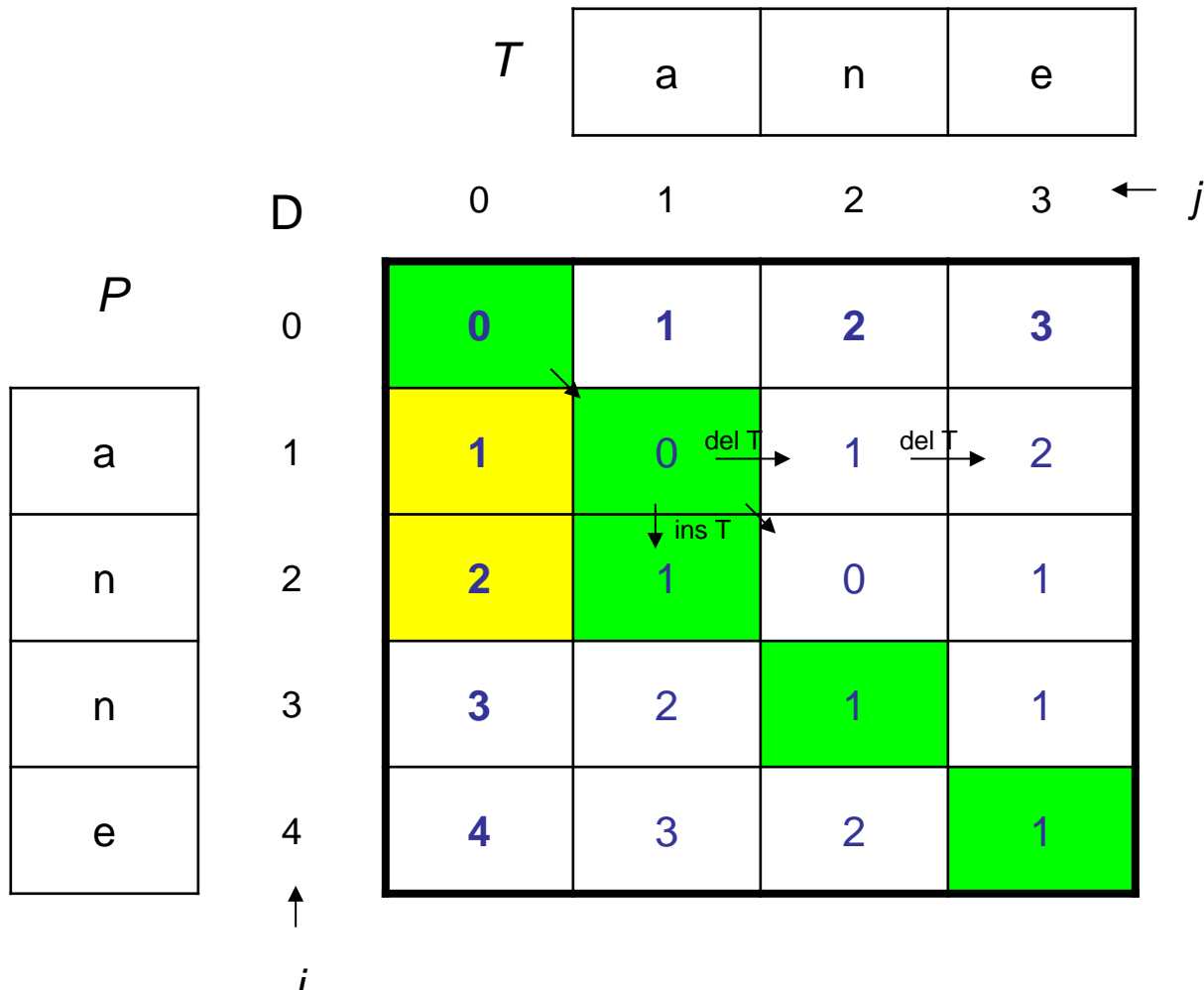
## «Top-Down» Dynamic Programming

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 solve an instance we first check whether it is already solved, and if so we only use the stored solution, otherwise we do recursive calls.
- 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" Method:

1. Start at the instance you want to solve, and ask recursively for the solution to the instances needed
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

## But:

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



# Another example: Optimal Matrix Multiplication

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:

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

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

# Optimal matrix multiplication 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

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

$A$  is a  $10 \times 100$  matrix,  $B$  is a  $100 \times 5$  matrix, and  $C$  is a  $5 \times 50$  matrix.

Computing  $D = (A \cdot B)$  costs 5,000 and gives a  $10 \times 5$  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  $100 \times 50$  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 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 paranthesizatoin 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 paranthesizatoin of the two parts, **we can sum their cost** and get the cost of the best paranthesizatoin given that we have to use this *outermost* partition.

Thus, to find the best paranthesizatoin 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 paranthesizatoin 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 paranthesizatoin 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  $M_i, M_{i+1}, \dots, M_j$  as  $m_{i,j}$ .**

# Optimal matrix multiplication 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  $m_{i,j}$  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  $m_{i,j}$  will be:

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

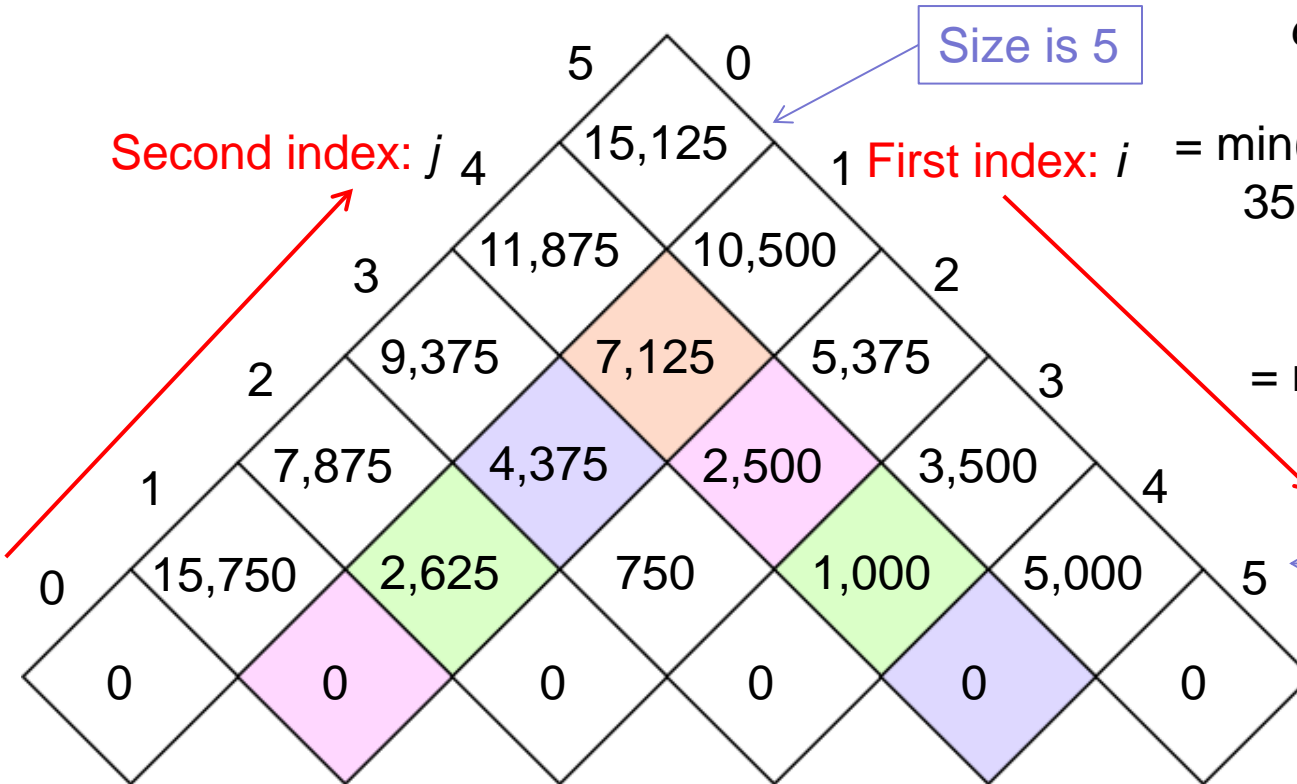
$$m_{i,i} = 0 \text{ when } 0 \leq i \leq n-1$$

Note that the values  $m_{k,l}$  that we need for computing  $m_{i,j}$  are all for *smaller* instances. That is:  $l - k < j - i$ .

# The table: Optimal matrix multiplication

$d$	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

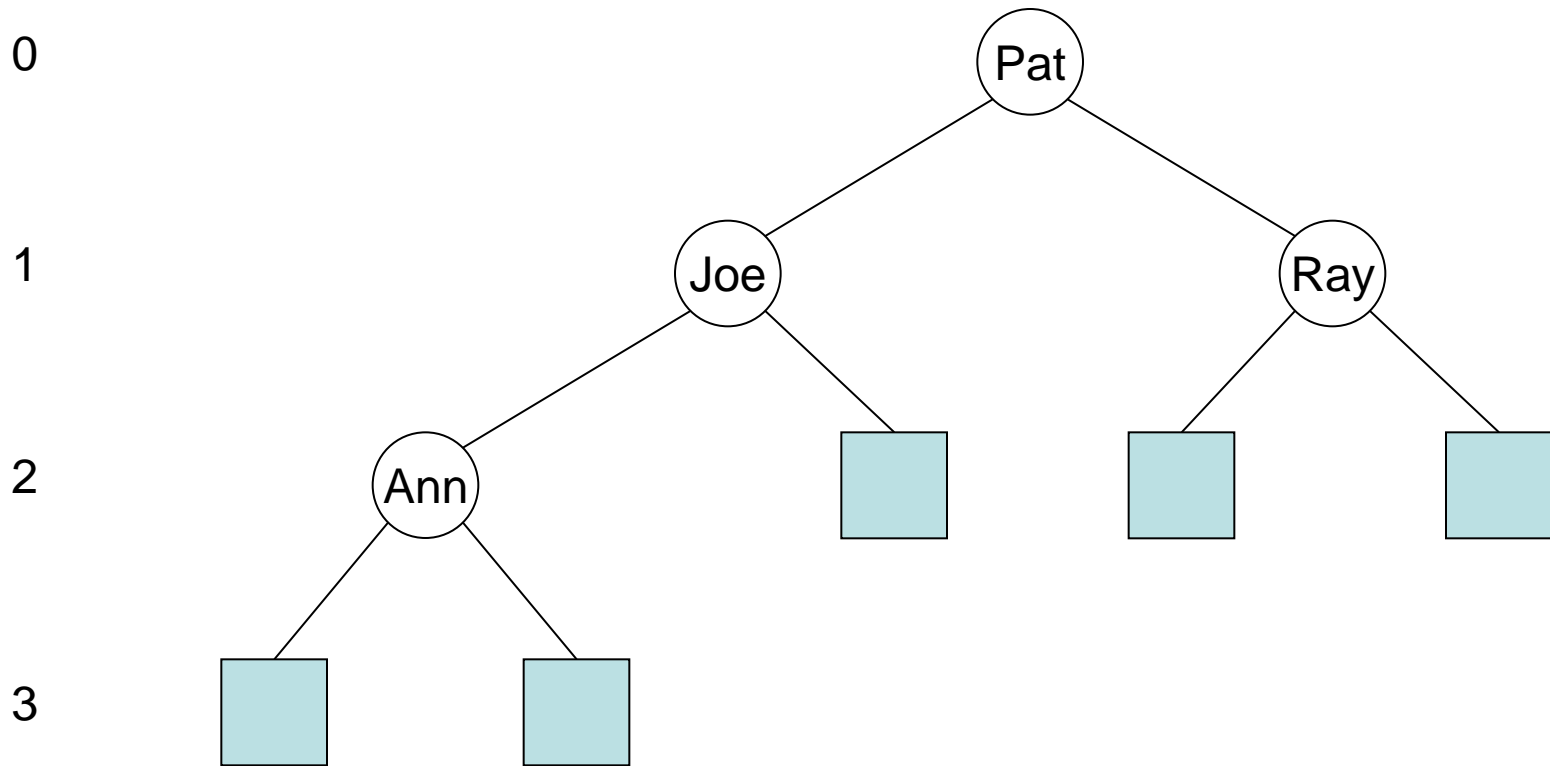
**Definition:** Size of the instance covering the interval from pos.  $i$  to pos.  $j$  is  $j - i$

# Program: Optimal matrix multiplication

```
function OptimalParens( d[0 : n - 1] )  
  for i ← 0 to n-1 do  
    m[i, i] ← 0  
  for diag ← 1 to n - 1 do  
    for i ← 0 to n - 1 - diag do  
      j ← i + diag  
      m[i, j] ← ∞ // Relative to the scalar values that can occur  
      for k ← i to j - 1 do  
        q ← m[i, k] + m[k + 1, j] + d[i] · d[k + 1] · d[j + 1]  
        if q < m[i, j] then  
          m[i, j] ← q  
          c[i,j] ← k  
        endif  
      endif  
    return m[0, n - 1]  
end OptimalParens
```

# Yet another example: Optimal search trees

(Not in the curriculum for 2016)



The sum of the  $p$ 's and  $q$ 's is 1

	Ann		Joe		Pat		Ray	
	$p_0$		$p_1$		$p_2$		$p_3$	
$q_0$		$q_1$		$q_2$		$q_3$		$q_4$
3	3	3	2	2	1	2	2	2

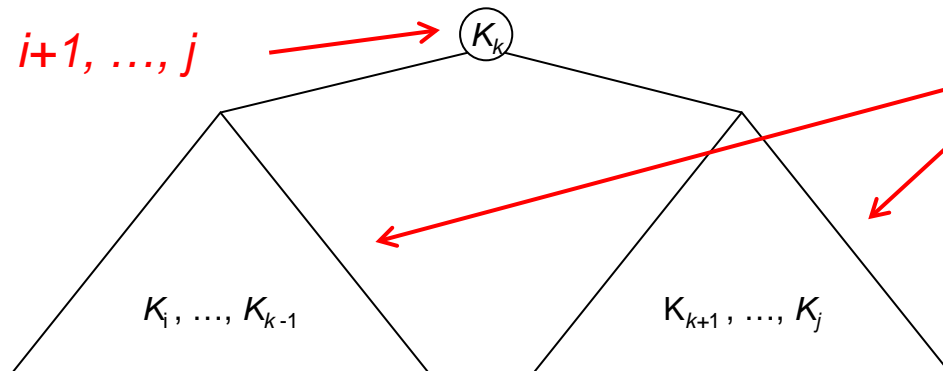
Average search time:  $3p_0 + 2p_1 + 1p_2 + 2p_3 + 3q_0 + 3q_1 + 2q_2 + 2q_3 + 2q_4$

# Optimal search trees

(Not in the curriculum for 2016)

- 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.

Try with  $k = i, i+1, \dots, j$

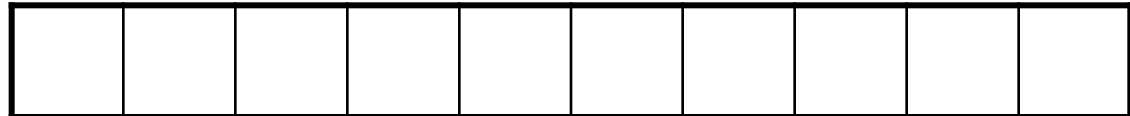
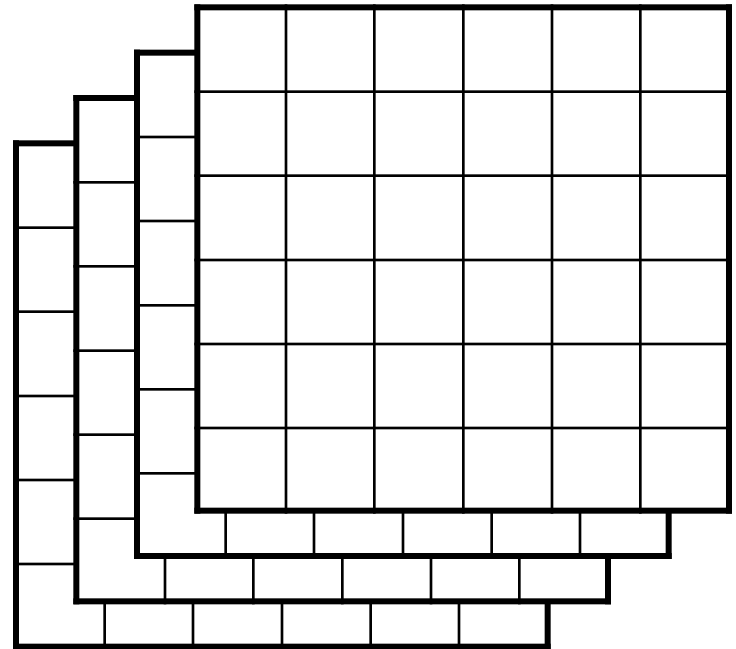
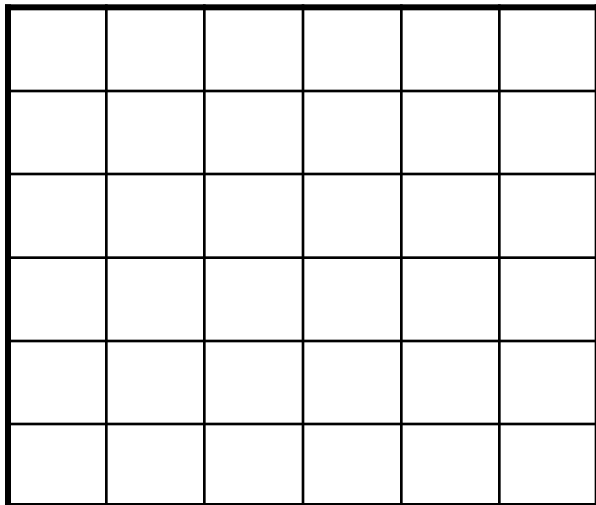
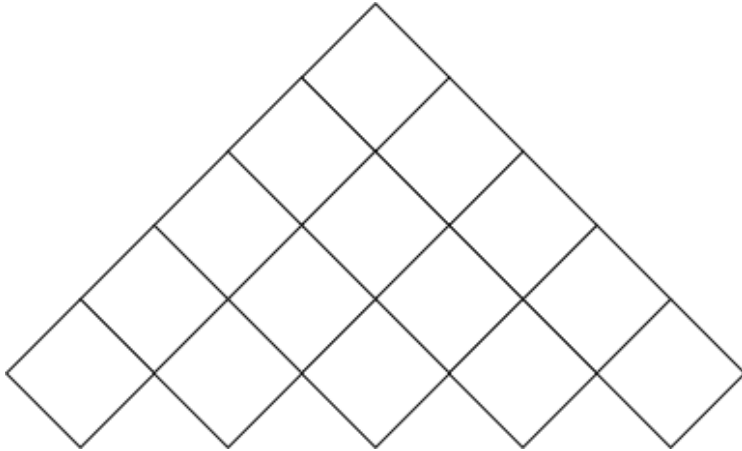


The optimal values and form for these subtrees are already computed, when we here try with different values  $K_k$  at the root



# Dynamic programming in general:

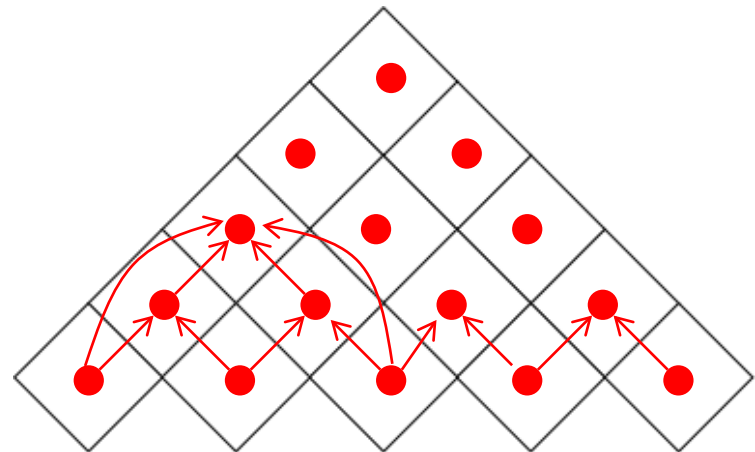
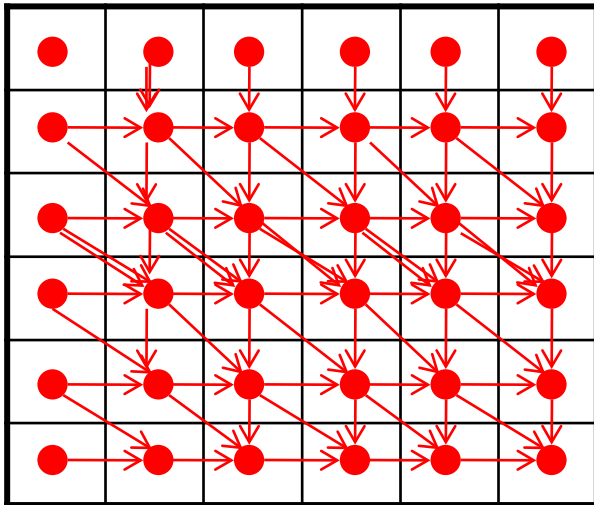
We fill in different types of tables «bottom up»  
(smallest instances first)



# 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 (see next slide).



# Dynamic Programming using memoization

## «Top-Down» dynamic programming (Memoization)

- A drawback with bottom up dynamic programming is that you might solve a lot of smaller instances whose answers are never used.
- We can instead do the computation recursively from the top, and store the (really needed) answers of the smaller instances in the same table as before. Then we can later find the answers in this table if we need the answer to the same instance once more.
- The reason we do not always use this technique is that recursion in itself can take a lot of time, so that a simple bottom up may be faster.
- For the recursive method to work, we need a flag «NotYetComputed» in each entry, and if this flag is set when we need that value, we compute it, and save the result (and turn off the flag, so the recursion from here will only be done once).
- The «NotYetComputed» flag must be set in all entries at the start of the algorithm.

# Dynamic programming using memoization

- 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 that. The important thing is that the smaller instances needed to solve a certain instance  $J$  is computed before we start solving  $J$ .
- 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.

At most the entries with a green dot will have to be computed

