INF4130: Dynamic Programming
September 2, 2014

- In the textbook: Ch. 9, and Section 20.5
- These slides were originally made by Petter Kristiansen, but are adjusted by Stein Krogdahl.
- The slides presented here have got a slightly different introduction than the one in the textbook
- This is because the way the textbook formulates the «principle of optimality» is not good.
- Thus some explanations for why the algorithms work will also be a little different on these 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.

The «DP-requirement» (DP=Dynamic Programming)

(this replaces the the «Principle of optimality»)

Assume we have:

- A problem *P* with many instances.
- An integer «*size*» for each instance, where the «simplest» instances have a small size, usually 0 or 1.
- For each instance *I* we have function

  $Combine_I(s(I_1), s(I_2), \dots , s(I_k))$

- where *k* may vary with the instance *I*, and
  - All of $I_1, I_2, \dots , I_k$ are smaller instances than *I*, and
  - *s(I)* is the solution to instance *I*

The DP-requirement then holds if

$Combine_I(s(I_1), s(I_2), \dots , s(I_k))$

gives the solution to the instance *I*. That is, it returns *s(I)*.

More on the DP requirement

- The problem P is usually an optimization-problem, so that *s(I)* is the best (in some sense) of all «feasible» solutions to *I*.
- The number of parameters to the $Combine_I$-function is either a constant, or a simple function of the instance *I*.
- The Combine-function can usually be expressed in a common form covering all instances of *P*
- And it can naturally be given as a program.

The DP requirement is often shown to be true for

a problem P, a *size* function, and a *Combine* function

by showing that

- if the value delivered by $Combine_I$ is not an optimal solution for an instance *I*,
- then at least one of the parameters to $Combine_I$ cannot be optimal for its (smaller than *I*) instance.

The Dynamic Programming algorithm

- Assume you have a problem *P*, and a function *Combine* that satesfies the DP requirement.

- Given an instance *I*, and make a table that has enough entriers to store the solution of all instances that will be required for solving *I* (must be found a separate argument)

- Initialize the table for instances with the smallest size z. The solutions to these are usually easy to find.

- Use the *Combine* function to first solve all relevant problems og size z+1, then all of size z+2, etc, and store the solutions in the corresponding table entry.

- You will then eventually find the solution of the instance you are interested in.

Example:

Searching for *similar* strings (Ch. 20.5)

This is relevant e.g. for research in genetics

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 opertions:

Substitution	One symbol in *T* is changed to another symbol.
Addition	A new symbol is inserted somwhere in *T*.
Removing	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!).*

Example.

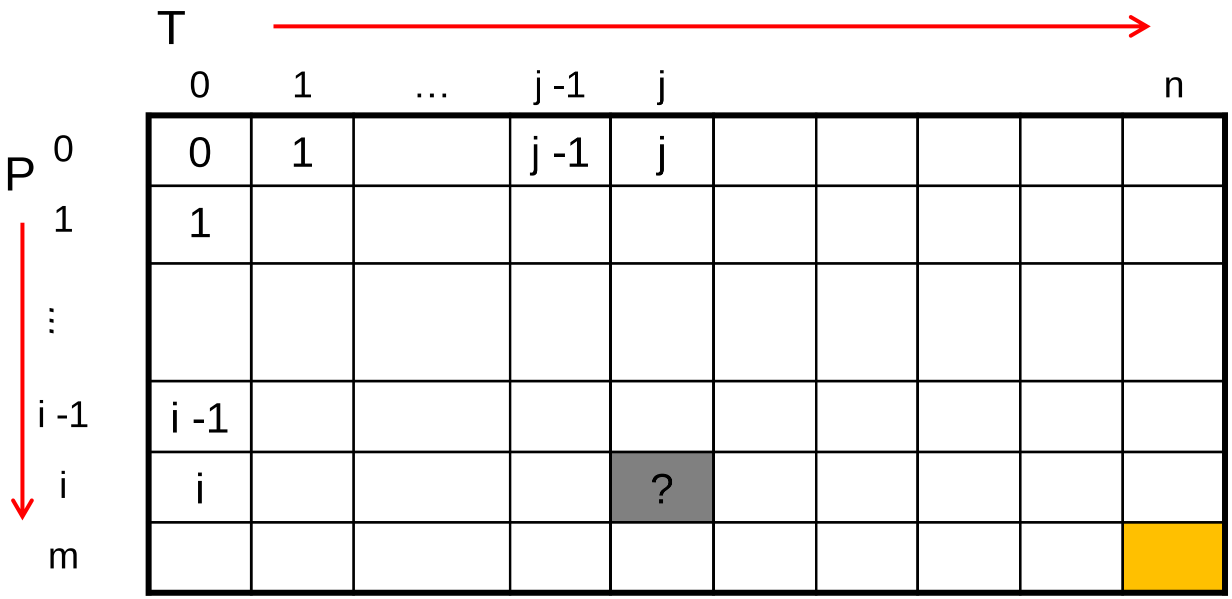
logarithm → alogarithm → algarithm → algorithm (Steps: +a, -o, a->o)
T *P*

Thus ED(”logarithm”, ”algorithm”) = 3 (as there are no shorter way)

Finding the Edit Distance

- Given two strings *T* and *P* of length m and n respectively.
- We want to find *ED*(*P, T*). The size of this instance is defined as *m+n.*
- The smaller problems we will solve to solve this, are to find the edit distance between *P* [1: *i*] and*T* [1: *j*] , 0<i<m and0< j<n. This problem has size i+j.
- We use a two-dimentional matrix *D[0:i, 0:m]* for storing the solutions for these smaller Instances. That is: *D* [*i*, *j*] = *ED*(*P* [1: *i*], *T* [1: *j*]).
- This is done *in the order «smaller instances first»*

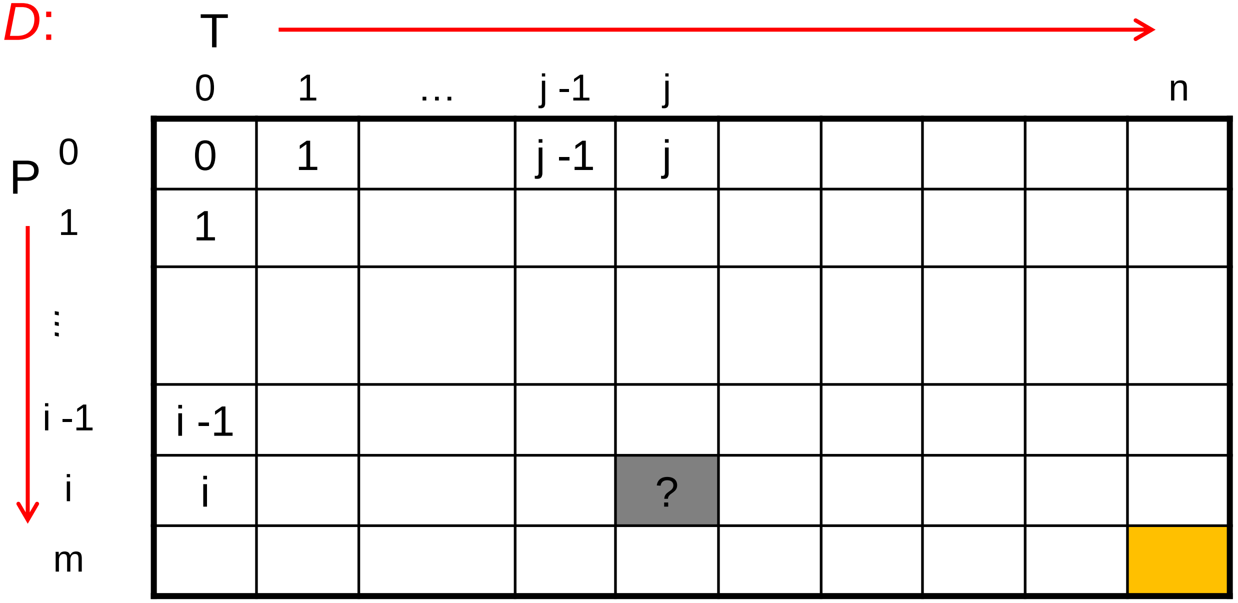
The matrix *D*:



	0	1	…	j -1	j					n
0	0	1		j -1	j					
1	1									
⋮										
i -1	i -1									
i	i				?					
m										

Finding the edit distance, initialization

The matrix *D*:



	0	1	…	j -1	j					n
0	0	1		j -1	j					
1	1									
⋮										
i -1	i -1									
i	i				?					
m										

- The problem with the smallest size occur when *i = j =0* (the size is 0). Then ED is obviously 0, as filled in above.

- We can also easily fill in for the cases where *i = 0* or *j = 0* (T or P is empty, that is: row 0 and line 0). Why can these be filled as indicated above?

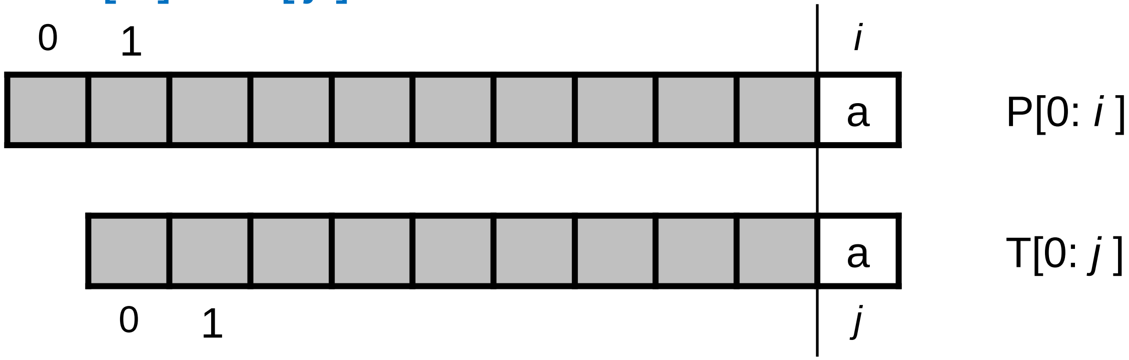
Finding the Edit Distance

It turns out that to find the value $D[i, j]$ we only need to look at the entries

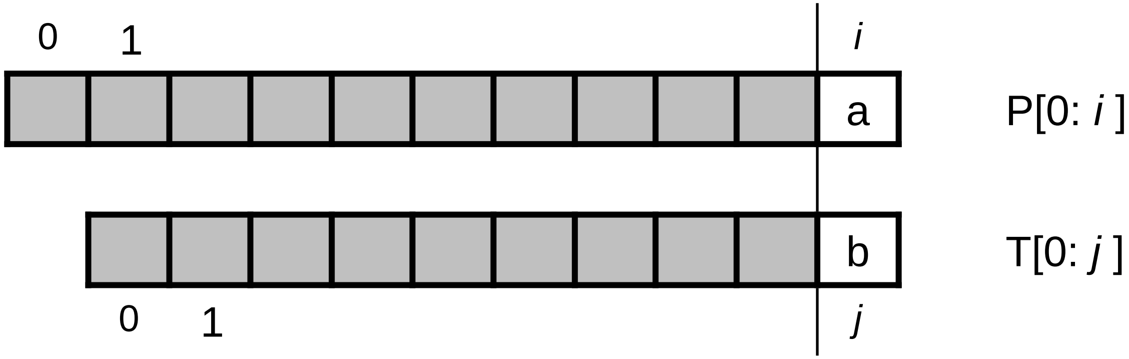
$D[i-1, j-1]$, $D[i, j-1]$, and $D[i-1, j]$

which all have smaller sizes than $D[i, j]$. We look at two cases:

Case 1: $P[i] = T[j]$

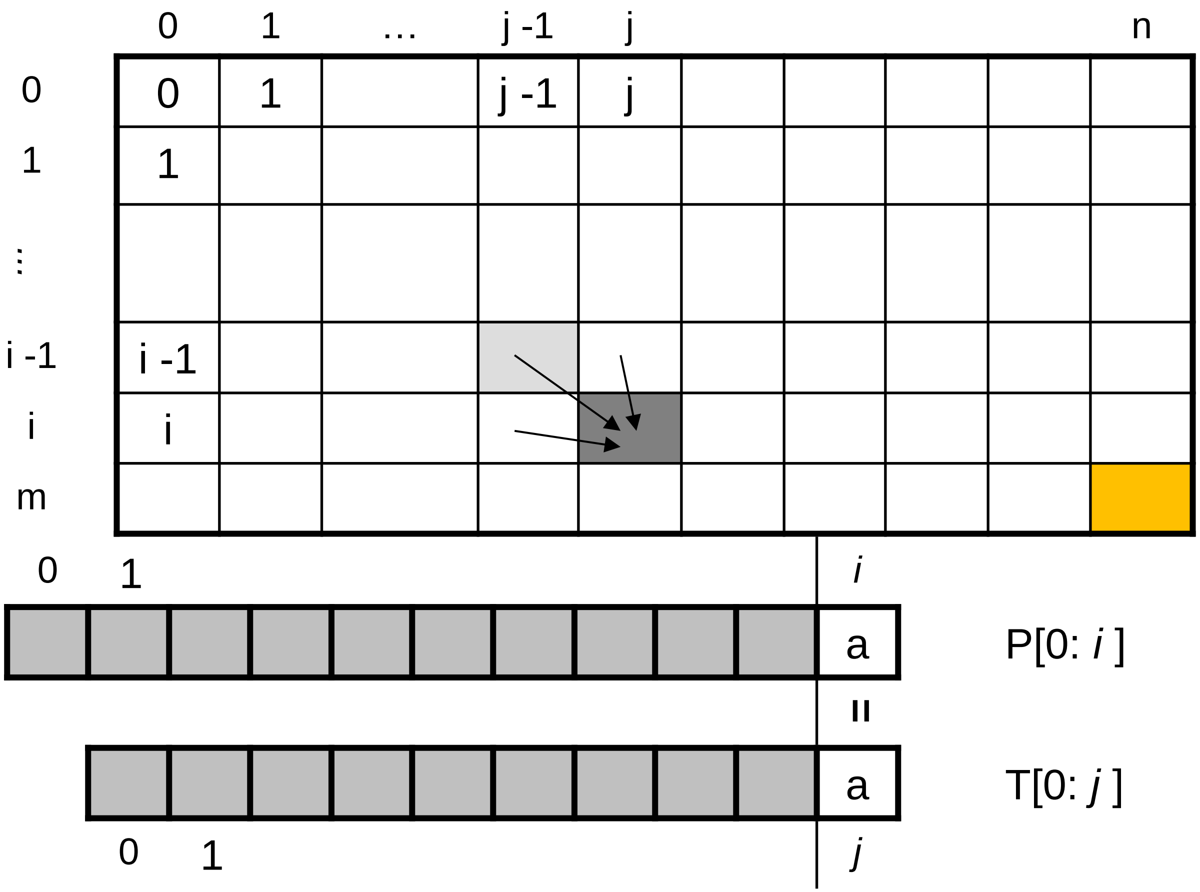


Case 2: $T[j]$ is *not* equal to $P[i]$.



Finding the Edit Distance

Case 1: If *P* [*i*] = *T* [*j*], then *D* [*i*, *j*] = *D* [*i* -1, *j* -1] (see figures below)

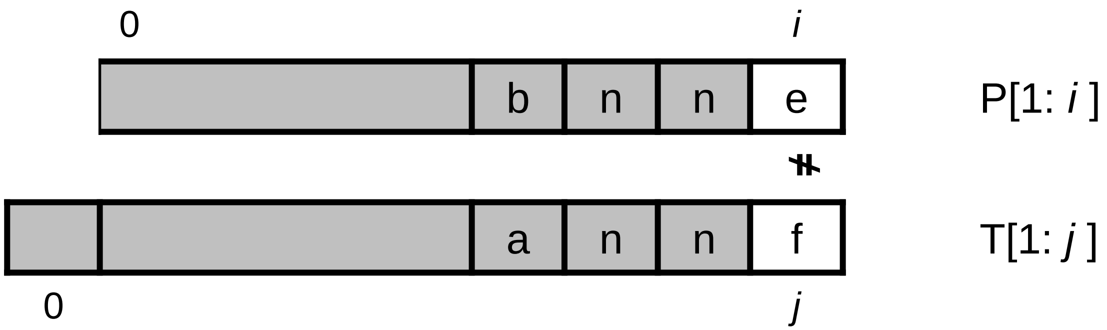


Why is it not possible to obtain a better ED?

Case 2: *T* [*j*] is *not* equal to *P* [*i*].

We choose the best of the following three possibilities:

A. Substitusjon – change *T* [*j*] to *P* [*i*]

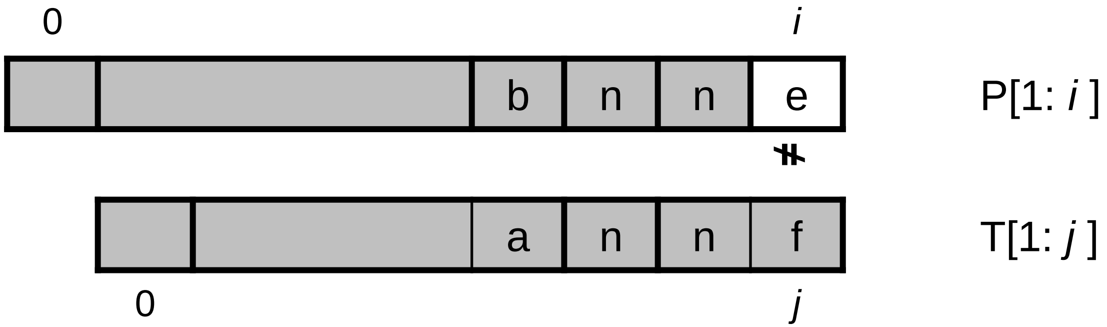


ED [*i*, *j*] would be

D [*i* -1, *j* -1] +1

(*ED* between the gray areas plus 1)

B. Addition of *T* [*j*] at the end of T

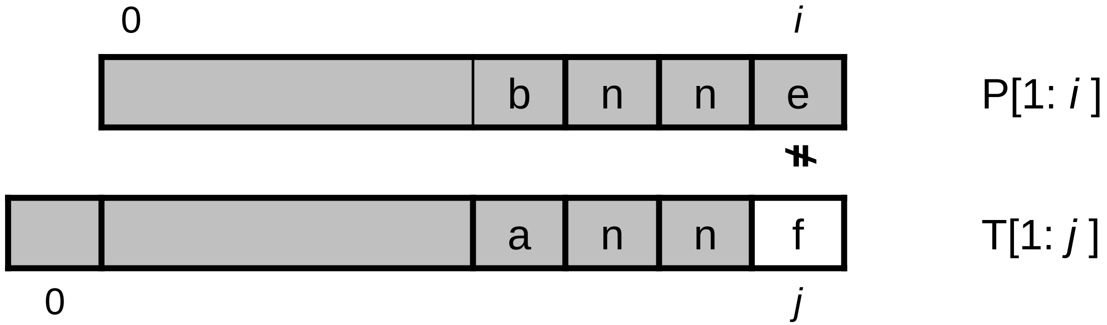


ED[*i*, *j*] would be

D[*i* -1, *j*] +1

(*ED* between the gray areas plus 1)

C. Remove *T* [*j*] from T.



ED[*i*, *j*] would be

D[*i* , *j* -1] + 1

(*ED* between the gray areas plus 1).

Computing edit distance.

(the *Combine* function has three parameters)

We propose the following expression for *D*[*i*, *j*]:

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

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

Proof for correctness:

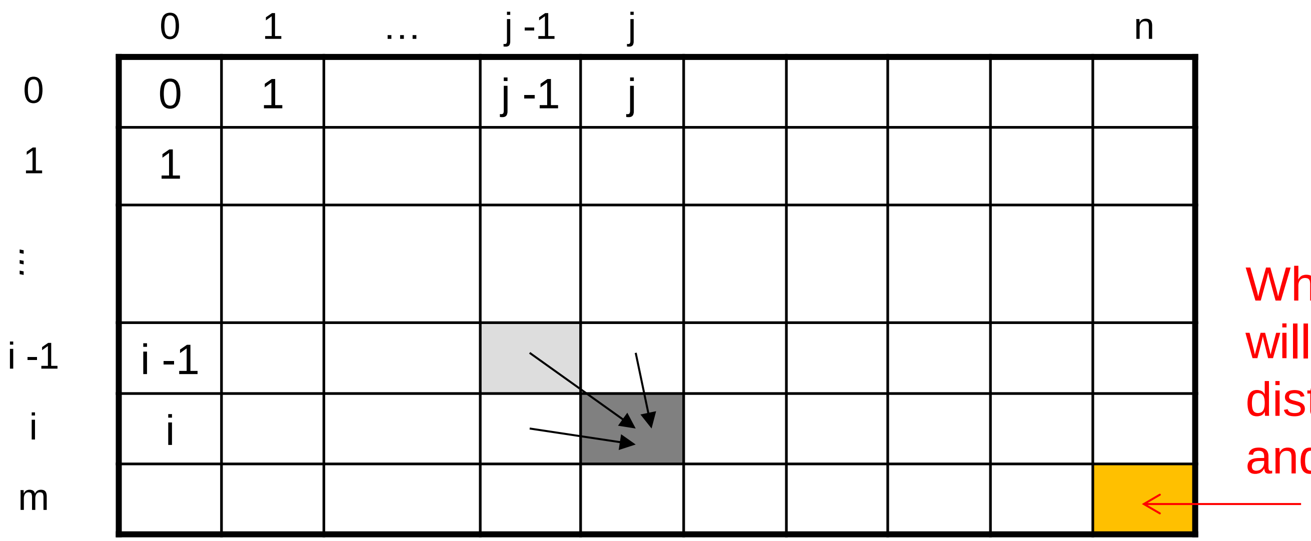
- Assume that
  1. *D[i-1,j-1], D[i-1,j], D[i,j-1]* are all optimal for the corresponding prolems
  2. D[i,j] computed the way given above is *not* optimal
- We then get a contradiction, as we from 2 can conclude that all of *D[i-1,j-1], D[i-1,j], D[i,j-1]* cannot be optimal. We can find a better value for at least one of them as the edit distance ED[i,j] cannot be larger than D[i,j] - 1

Computing edit distance

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

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

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



	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*[*n*,*m*]

We will fill in the entries of the matrix in the order from smaller to larger sizes, starting with the initialisation given above.

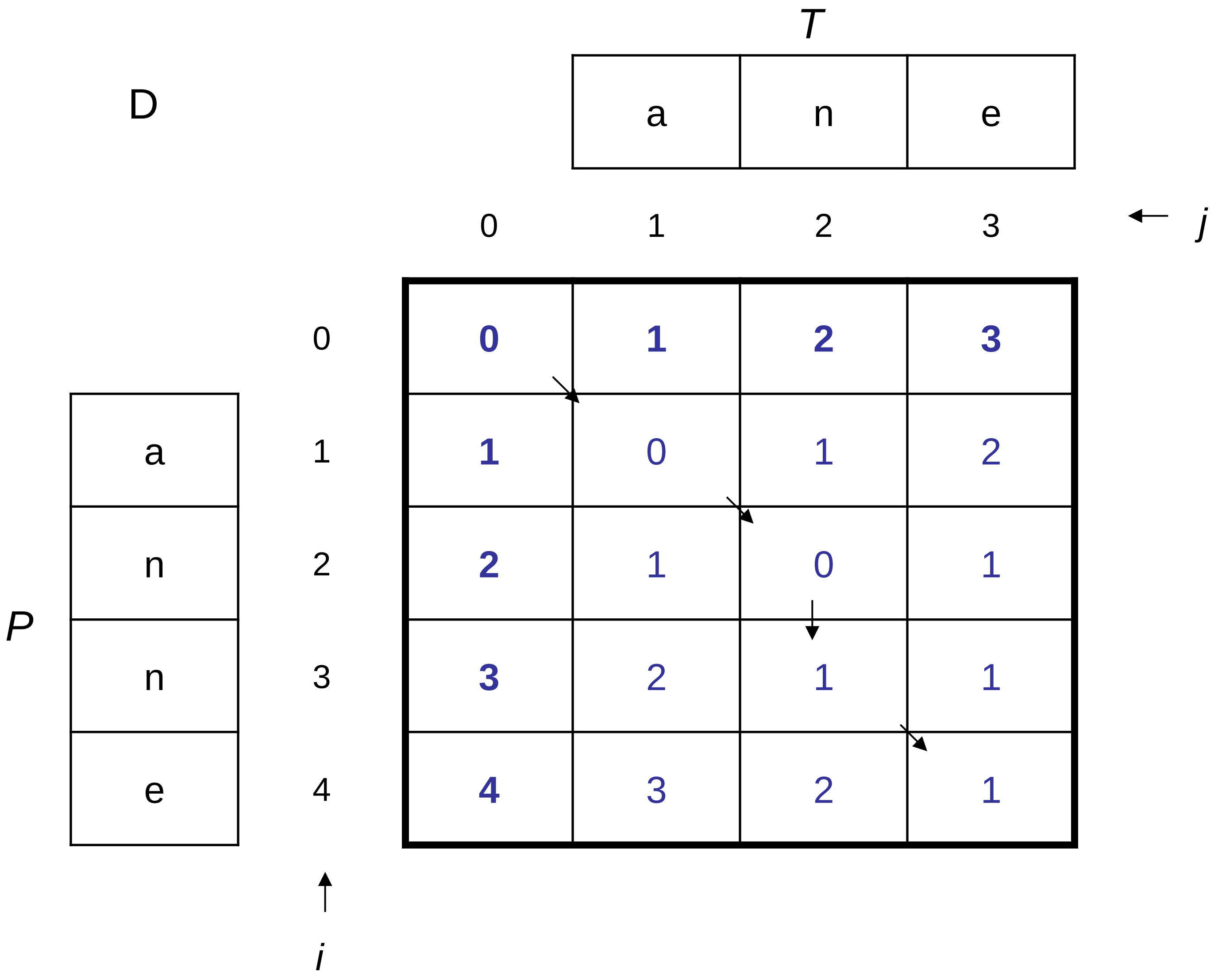
Computing edit distance

```
function EditDistance ( $P[1:n]$, $T[1:m]$ )
    for $i \leftarrow 0$ to $n$ do $D[i, 0] \leftarrow i$ endfor
    for $j \leftarrow 1$ to $m$ do $D[0, j] \leftarrow j$ endfor

    for $i \leftarrow 1$ to $n$ do
        for $j \leftarrow 1$ to $m$ 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( $D[n, m]$ )
end EditDistance
```

Note that this algorithm does *not* go through the instances strictly in the order from smaller to larger ones. In fact, after the initialization we use the following order of the pairs (i, j):
(1,1) (1,2) … (1,m) (2,1) (2,2) … (2,m) (3,1) (3,2) … (n,m)
This is OK as also this order ensures that the smaller instances are solved before they are needed to solve a larger instance. An order strictly following increasing size would also work fine, but is slightly more complex to program (following diagonals).

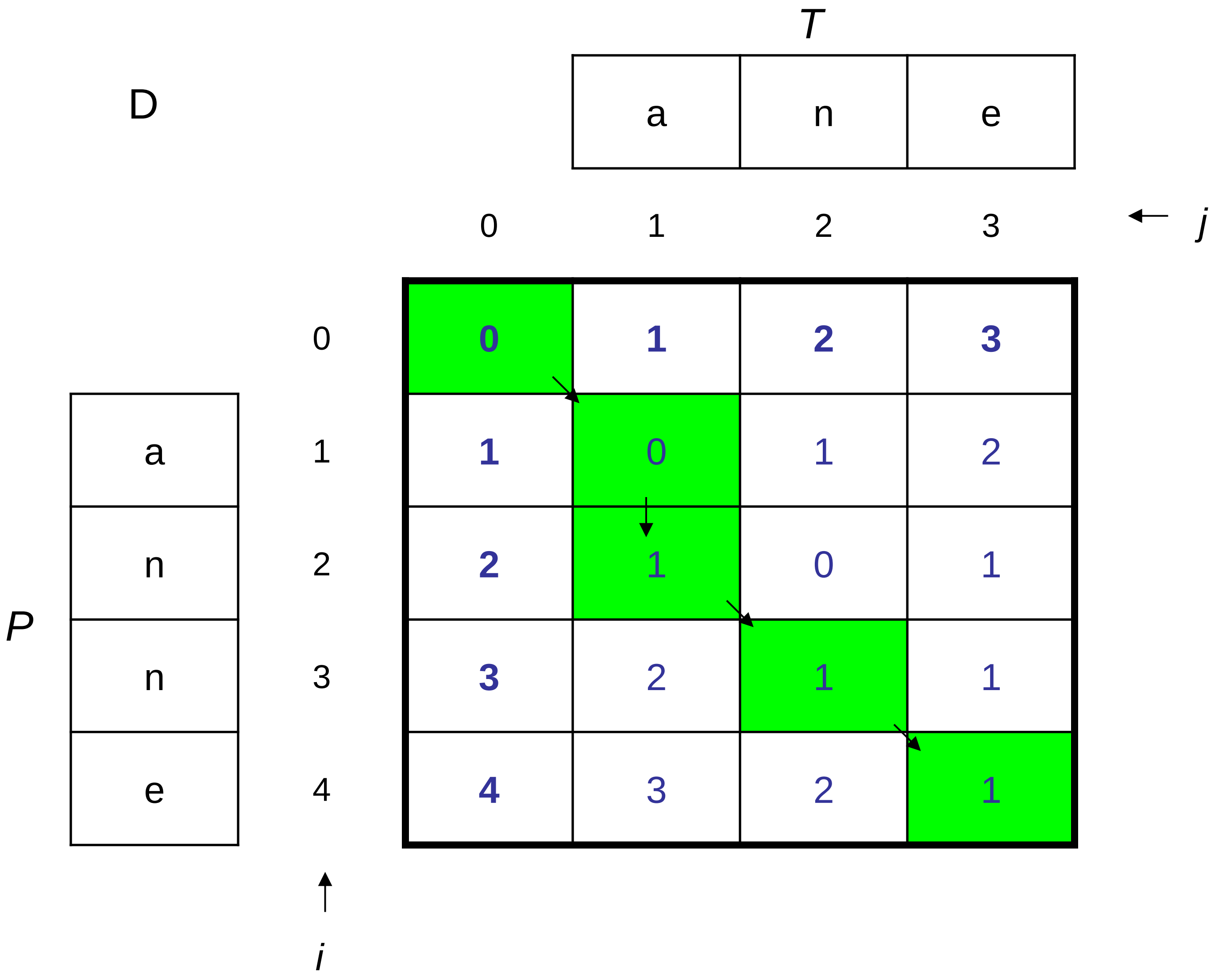
Example



			a	n	e
		0	1	2	3
	0	**0**	**1**	**2**	**3**
a	1	**1**	0	1	2
n	2	**2**	1	0	1
n	3	**3**	2	1	1
e	4	**4**	3	2	1

Computing edit distance

Example



D

T

			a	n	e
		0	1	2	3
	0	0	1	2	3
a	1	1	0	1	2
n	2	2	1	0	1
n	3	3	2	1	1
e	4	4	3	2	1

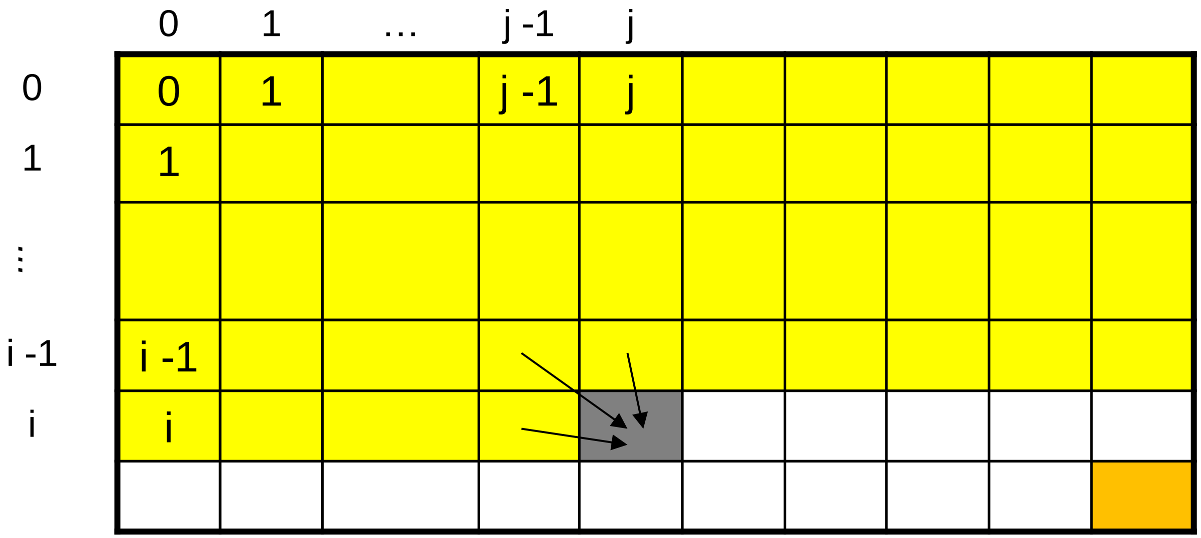
Summing up dynamic programming - 1

- Dynamic programming is typically used to solve optimazation problems. That is, problems that can have a number of «feasible» solutions, but where we want to find the «best» – by optimizing the value of a given *objective function.*

- Each instance of the problem must have an integer *size*. Typically the smallest (or simplest) instances have size 0 or 1, that can easily be solved.

- For each problem instance I of size *n* there is a set of instances I_1, I_2, … ,I_k, all with sizes less that n, so that we can find an (optimal) solution to I if we know the (optimal) solution of the I_i-problems.

Example:



The values of the yellow area is computed when the gray value is to be computed

Summing up dynamic programming

In the textbook (page 265)

The solution to the instance I is called S, and that of each I_i is called S_i. The way to find S from the S_i is written:

$$S = \text{Combine}(S_1, S_2, \ldots, S_m)$$

For our example problem:

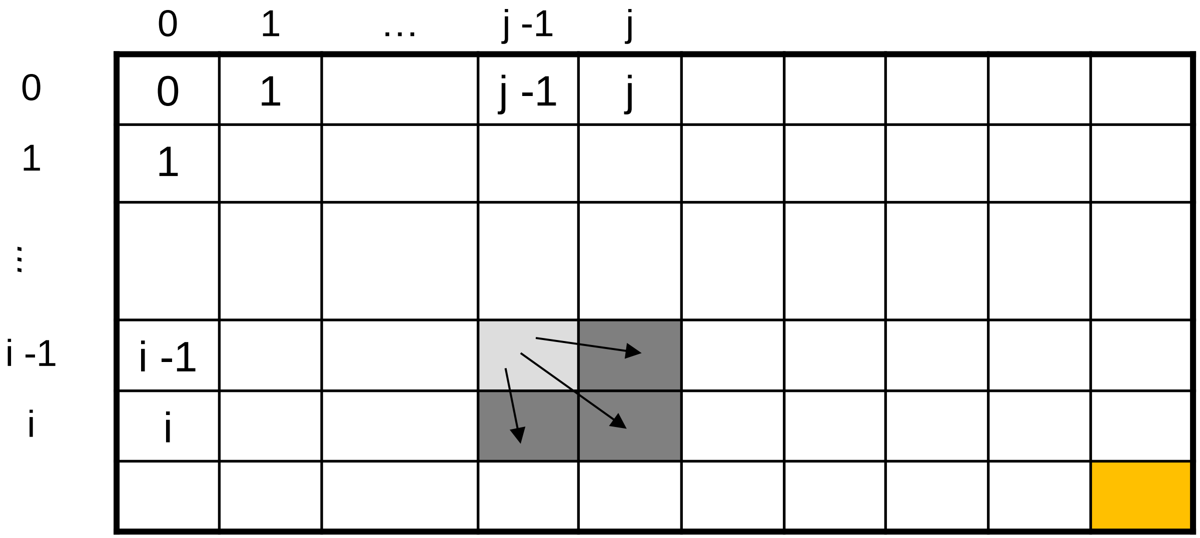
$$D[i,j] = \begin{cases} D[i-1,j-1] & \text{if} \quad P[i] = T[j] \\ \min\{\ \underbrace{D[i-1,j-1]+1}_{\text{substitusjon}},\ \underbrace{D[i-1,j]+1}_{\substack{\text{tillegg i } T \\ \text{sletting i } P}},\ \underbrace{D[i,j-1]+1}_{\text{sletting i } T}\ \} & \text{otherwise} \end{cases}$$

$$D[0,0] = 0, \quad D[i,0] = D[0,i] = i. \qquad \text{initialization}$$

Summing up dynamic programming

- Dynamic programming is useful if the total number of smaller instances (recursively) needed to solve an instance A is so small that the answer to all of them can be stored in a table.

- For dynamic programming to be useful, the solution to a given instance B will be used in a number of problems A with size larger than that of B. The main trick is to store solutions for later use.

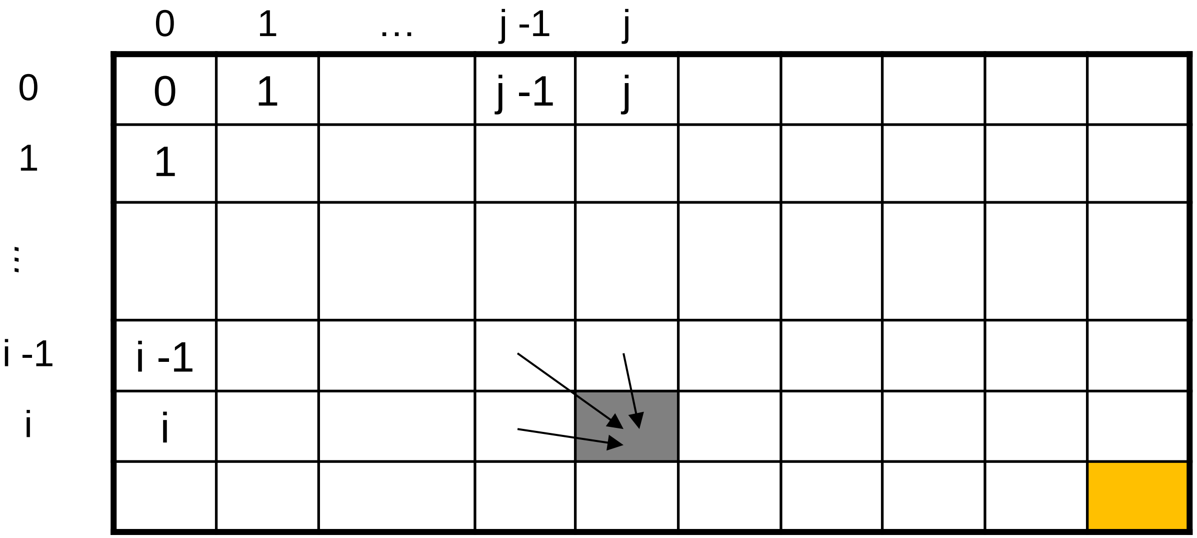


	0	1	…	j -1	j					
0	0	1		j -1	j					
1	1									
⋮										
i -1	i -1									
i	i									
										🟨

When to use dynamic programming?

- Thus, if we compute and store (in a table of a suitable format), the solutions to all relevant instances of a given size before looking at instances of larger sizes, we will always know the arguments to the Combine-function when we need them for computing the solution of an instance of larger size.

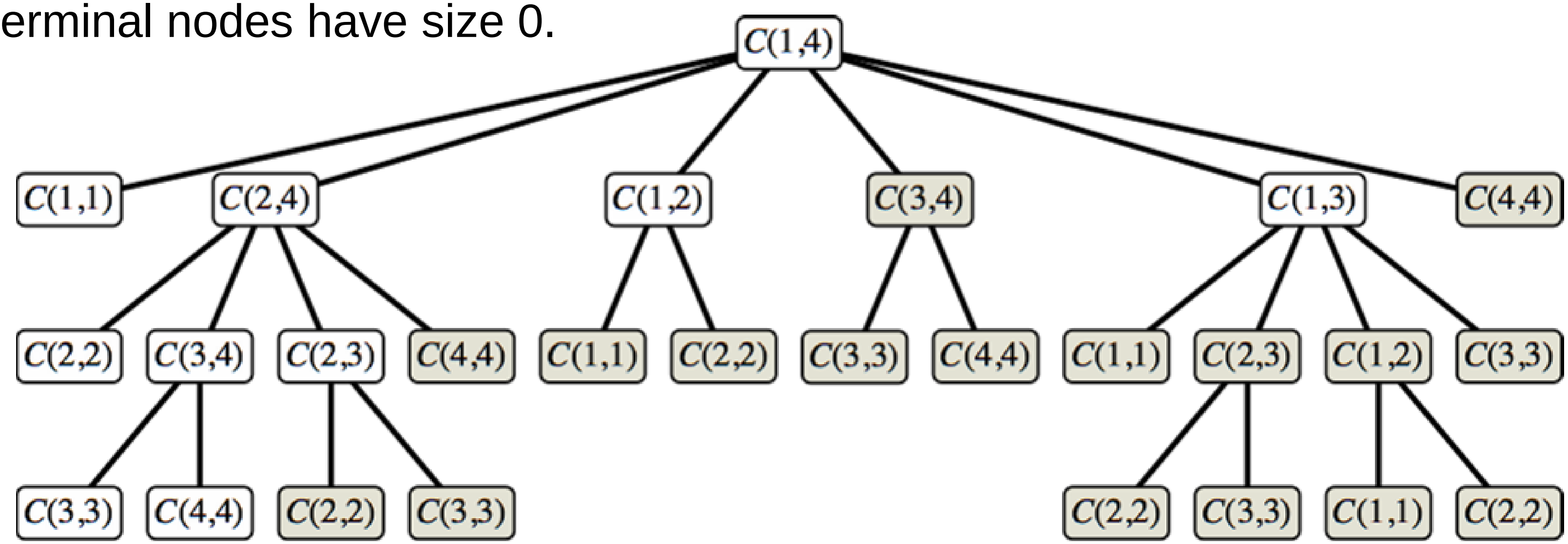
- We start by solving the smallest instances, and then look at larger and larger instances (all the time storing the solutions).



	0	1	…	j -1	j					
0	0	1		j -1	j					
1	1									
⋮										
i -1	i -1									
i	i									

Another view

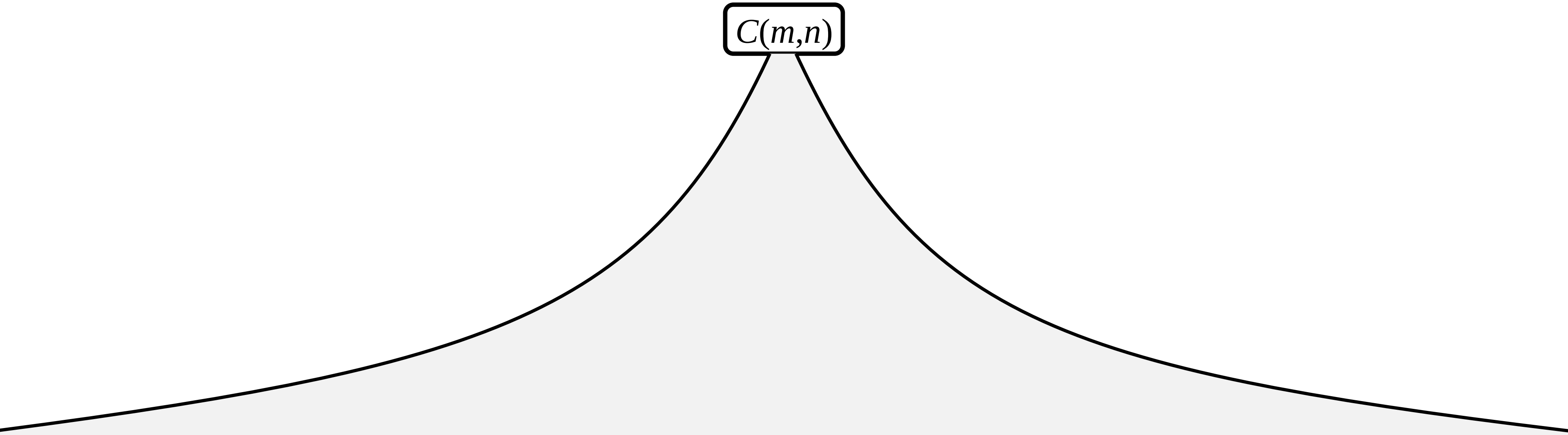
- As indicated on the previous slide, Dynamic Programming is useful if the solution to a certain instance is used in the solution of many instances of larger size.
- In the problem C below, an instance is given by some data (e.g two strings) and by two intergers *i* and *j.* The corresponding instance is written *C*(*i*, *j*). Thus the solutions to the instances can be stored in a two-dimentional table with dimensions *i* and *j*.
- The size of an instanstance C(i, j) is j – i
- Below, the children of a node *N* indicate the instances of which we need the solution for computing the solution to *N*.
- Note that the solution to many instances, e.g. C(3,4), is used multiple times. This is required to make DP a preferable alternative! Also note that all terminal nodes have size 0.



Number of smaller solutions needed

- If the number of solutions to the different smaller instances that are needed to find the soluton to a certain instance is very big (e.g. exponential in the size of the instance), then the resulting DP algoritm 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 it is rather small.



Sketch indicating the situation when the solution of one instance needs the solution of many *different* smaller instances.

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 the solutions are stored in a table.
- Works best when the answers to smaller instances are needed by many larger instances.

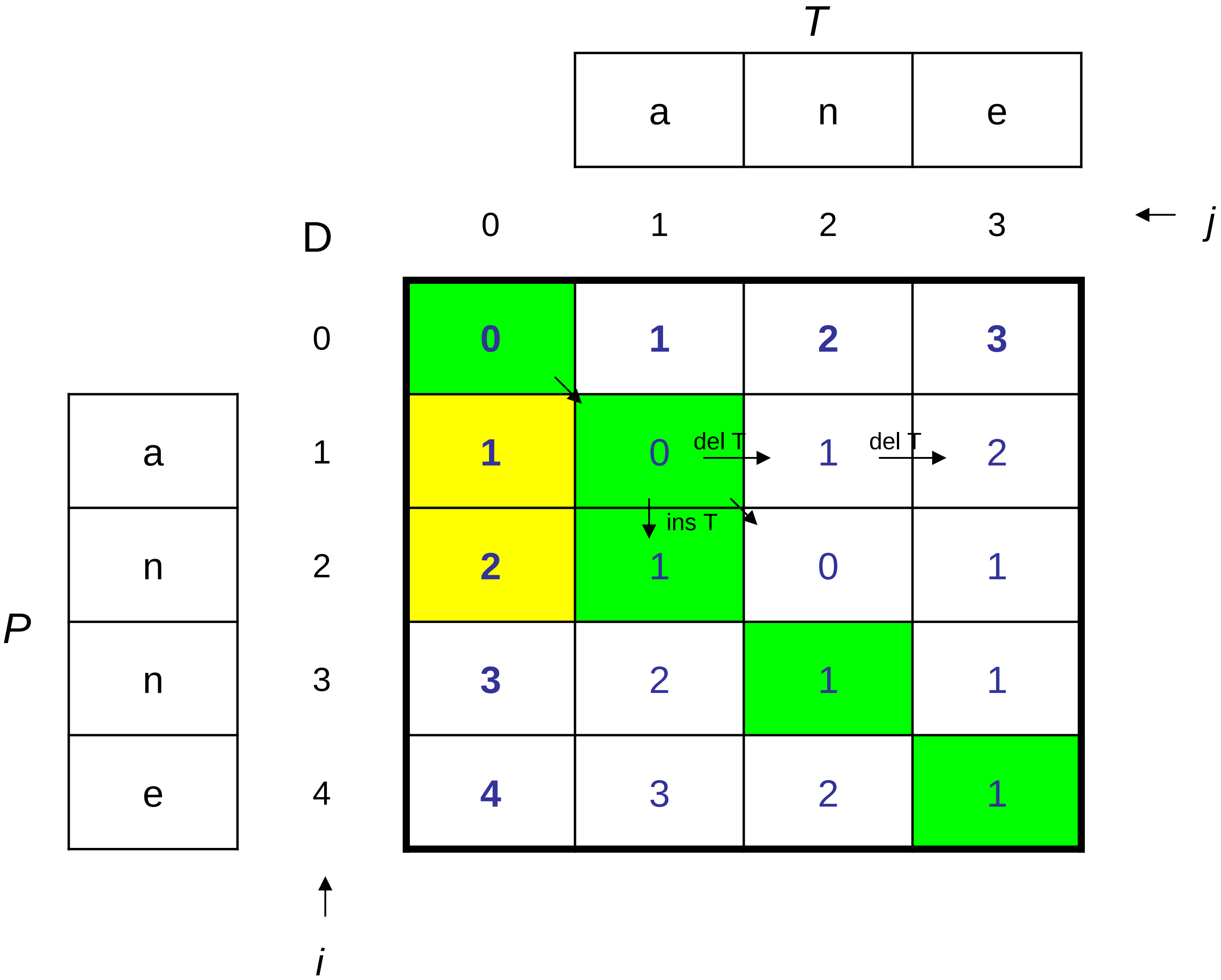
«Top-Down» Dynamic Programming («Memoization», more later**)**

- A drawback with (traditional) dynamic programming is that one usually solve a number of smaller instances that turns out not to be needeed for the actual (larger) instance you originally wanted to solve.
- We can instead start at this actual instance we want to solve, and do the computation top-down (usually recursively), and put all computed solutions into the same table as above (see later slides).
- 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”

Benefit: You only have to compute the needed table entries
But managing the recursive calls take some extra time



T

			a	n	e
	D	0	1	2	3
	0	0	1	2	3
a	1	1	0	1	2
n	2	2	1	0	1
n	3	3	2	1	1
e	4	4	3	2	1

New 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 · B) · C = A · (B · C)

(but not symmetric, since A · B generally is different from B · A)

Thus, one can do the multiplications in different orders. E.g., with four matrices it can be done in the following five ways:

$(M_0 \cdot (M_1 \cdot (M_2 \cdot M_3)))$
$(M_0 \cdot ((M_1 \cdot M_2) \cdot M_3))$
$((M_0 \cdot M_1) \cdot (M_2 \cdot M_3))$
$((M_0 \cdot (M_1 \cdot M_2)) \cdot M_3)$
$(((M_0 \cdot M_1) \cdot M_2) \cdot M_3)$

The cost (the number of simple (scalar) multiplications) of these will usually vary a lot for the differnt 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 dimentions:

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 × 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 · (B · 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, …, 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 parenthesizaton is a partition into two parts: $(M_0 \cdot M_1 \cdot \ldots \cdot M_k) \cdot (M_{k+1} \cdot M_{k+2} \cdot \ldots \cdot M_{n-1})$
If we know the best parenthesizaton of the two parts, we can sum their cost and get the cost of the best parenthesizaton given that we have to use this *outermost* partition.

Thus, to find the best parenthesizaton of M_0, M_1, …, 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 parenthesizaton of all (or a lot of) instances of smaller sizes.

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

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

We will refer to the lowest possible cost for M_i, M_{i+1}, …, M_j as $m_{i,j}$.

Optimal matrix multiplication - 4

Let d_0, d_1, …, d_n be the dimensiones of the matrices M_0, M_1, …,$M_{n\text{-}1}$, 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 parenthesizaton of M_{i}, $M_{\mathrm{i+1}}$, …, M_j.
Thus the value we are interested in is $m_{0,n\text{-}1}$

The recursive fomula for $m_{i,j}$ will be:

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

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

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

Example: Optimal matrix multiplication

d	30	35	15	5	10	20	25

Example

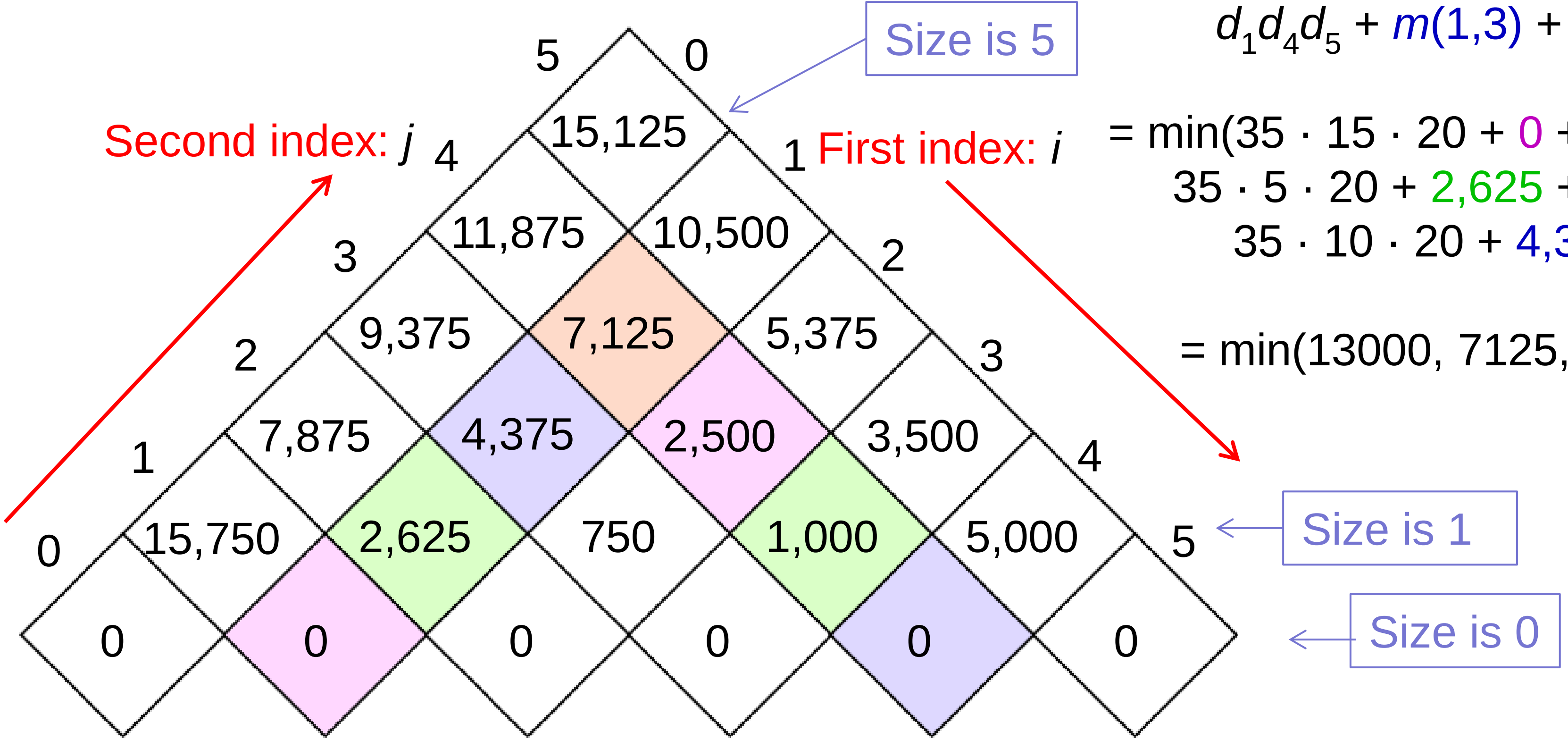
$$m_{1,4} = \min(d_1 d_2 d_5 + m(1,1) + m(2,4),\ d_1 d_3 d_5 + m(1,2) + m(3,4),\ d_1 d_4 d_5 + m(1,3) + m(4,4))$$

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

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

$$=7125$$

The values $m_{i,j}$:



Second index: *j*

First index: *i*

Size is 5

Size is 1

Size is 0

15,125

11,875 10,500

9,375 7,125 5,375

7,875 4,375 2,500 3,500

15,750 2,625 750 1,000 5,000

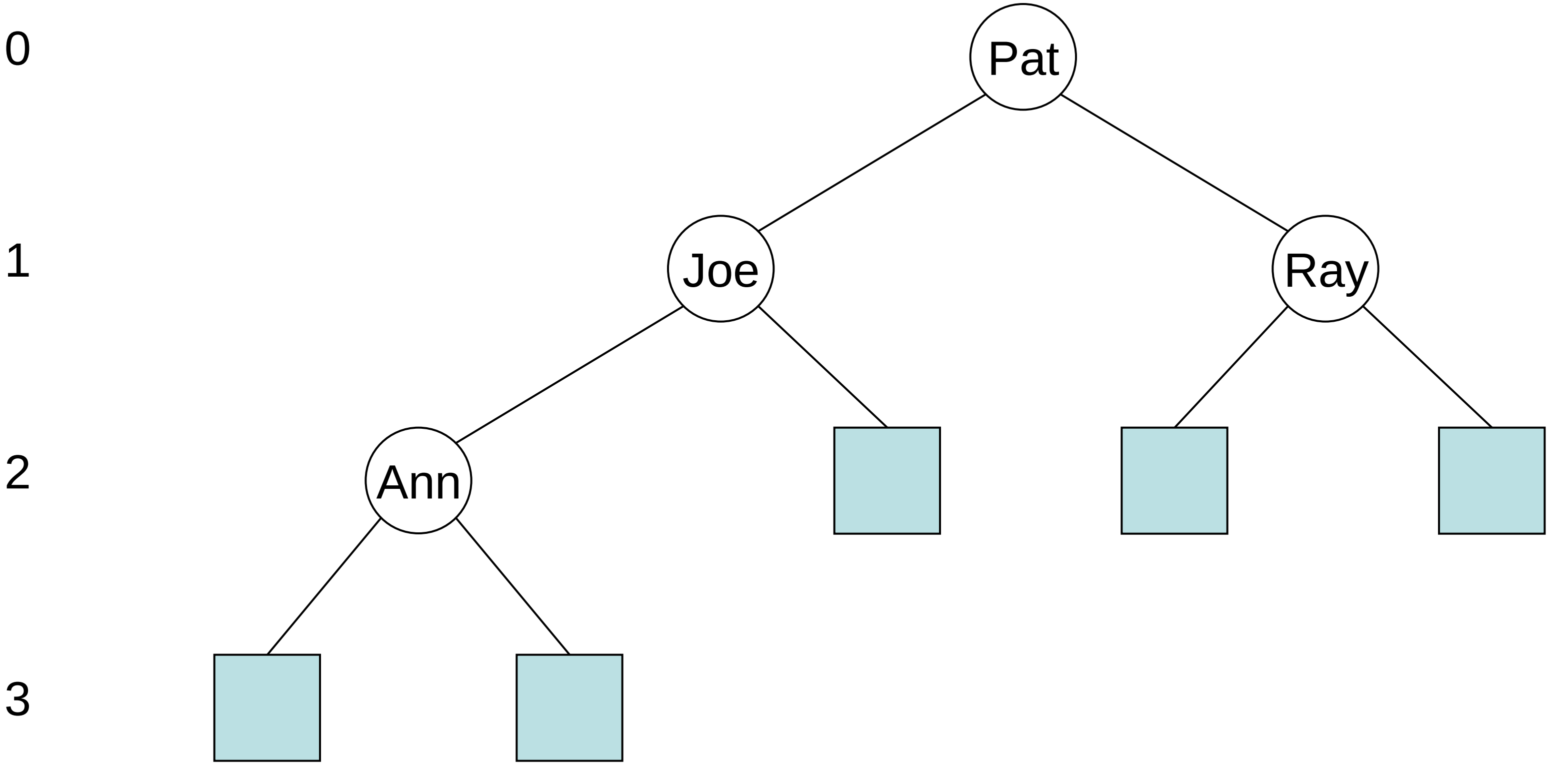
0 0 0 0 0 0

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
    return m[0, n – 1]
end OptimalParens
```

Yet another example: Optimal search trees

(Not in the curriculum for 2014)



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

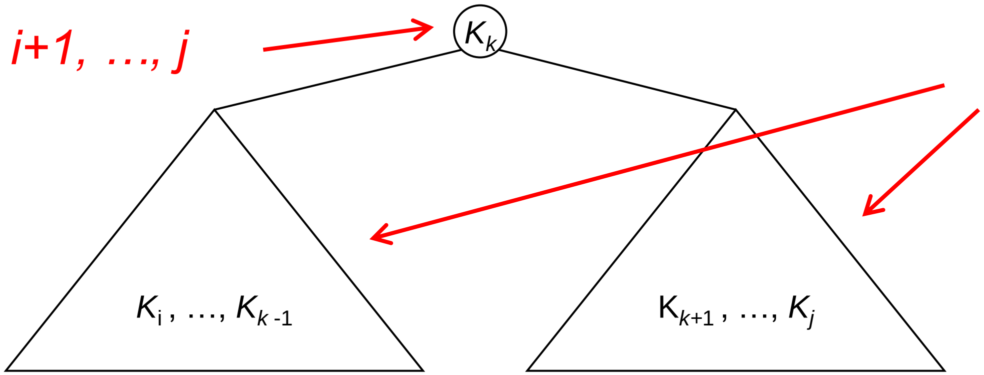
Average search time: $3p_0 + 2p_1 + 1p_2 + 2p_3 + 3q_0 + 3q_1 + 2q_2 + 2q_3 + 2q_4$

Optimal search trees

(Not in the curriculum for 2014)

- To get a managable 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 seach 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 intex *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 the an interval with values K_{i}, …, K_j we can simply try with each of the values K_i, …, K_j as root, and use the best subtrees in each of these cases (which are already computed).
- To compute the cost of the subtrees is slightly more complicated than in the matrix case, but is no problem.

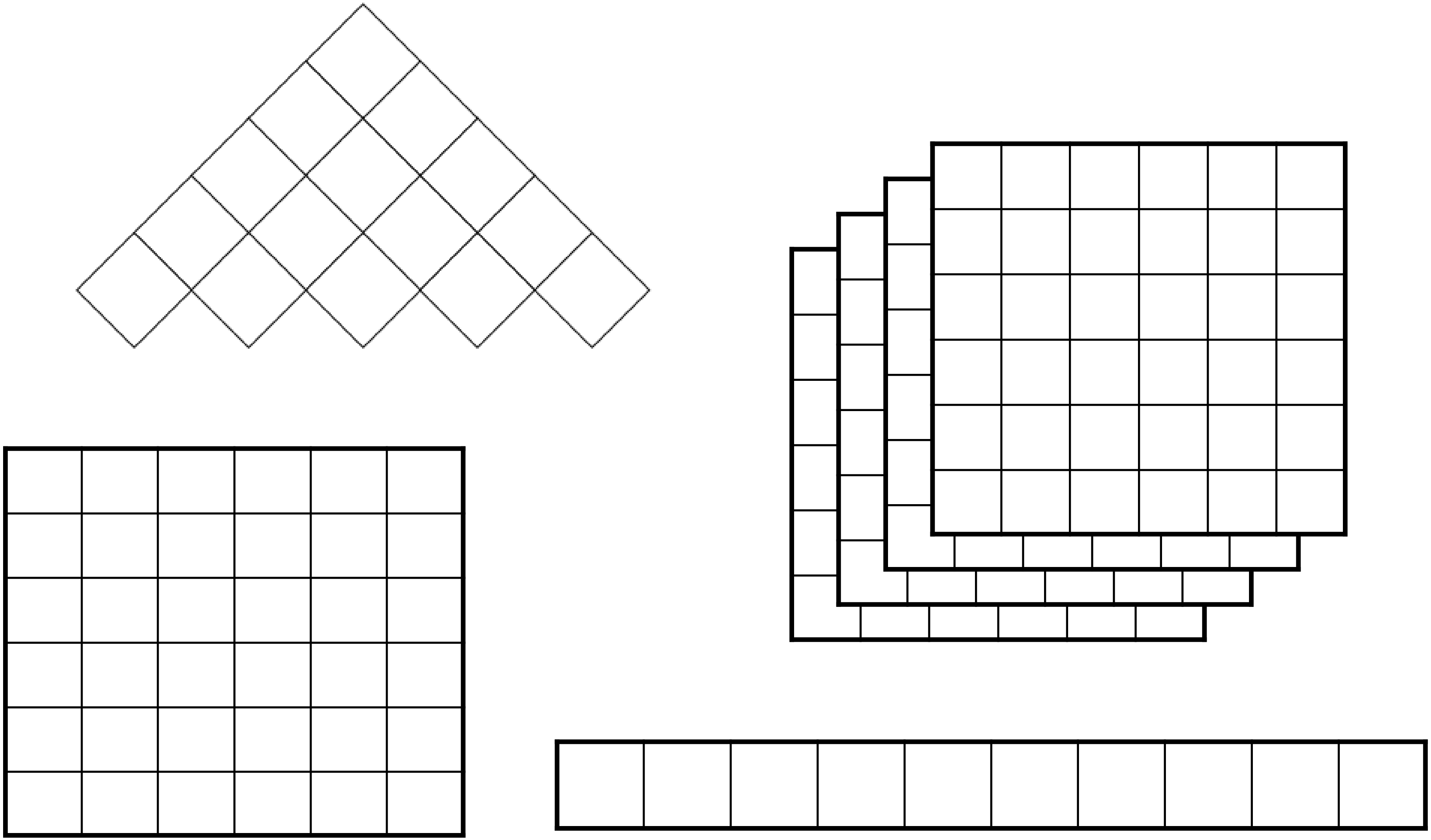
Try with *k*= *i, i*+1*, …, j*



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 differnt types of tables «bottom up»
(smallet instances first)



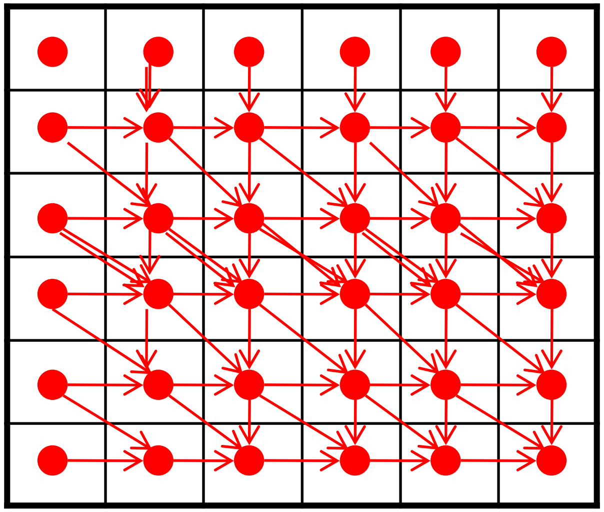
Dynamic programming

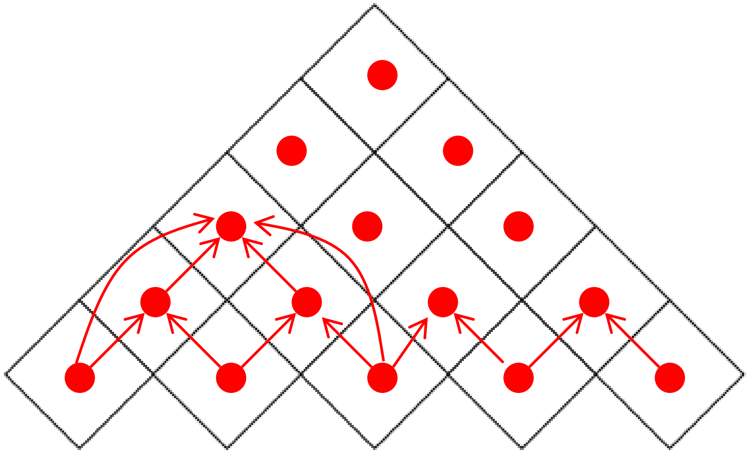
Filling in the tables

- 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