INF4130: Dynamic Programming

Slides to the lecture Sept. 13, 2018.

- In the textbook: Ch. 9, and Section 20.5
  - The discussion of this example in Sec. 20.5 relies on Ch. 9, and is therefore rather short in then textbook.
- 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, and the formulation of the «principle of optimality» is not good (it should be the other way around)
  - Thus, some explanations for why the algorithms work will somewhere be different from that in the textbook.
- And the curriculum is the version used in 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.

In a moment we shall look at the following 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: u t t x v

T: b s u t t v r t o f i g u t t v x l 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

Questions:

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

We'll soon get back to this topic, and:

- It is highly connected to a problem called Shortest Edit Distance.
- The last step for solving Approximate String Matching problem will be studied as an assignment next week.

But first, some simpler examples

1. The Fibonacci numbers

Definition: fib(0)=0 fib(1)= 1 For n ≥ 2: fib(n) = fib(n-1) + fib(n-2)

n:	0	1	2	3	4	5	6	7	8	9	10	11	12	. . .
fib(n):	0	1	1	2	3	5	8	13	21	34	55	89	144	. . .

The normal way to compute them (e.g. for sequential output) is:

Always remember the two last values in «prev» and «prevprev»

Initialize: {prevprev = 0; prev= 1; n = 1} and repeat the following:

{ cur = prev + prevprev; n = n+1; ouput(n, cur); prevprev = prev; prev = cur; }

If you want to store the sequence in an array «fib[M]» then do:

fib[0] = 0; fib[1] = 1; for n= (2 to M-1) { fib[n] = fib[n-1] + fib[n-2]; }

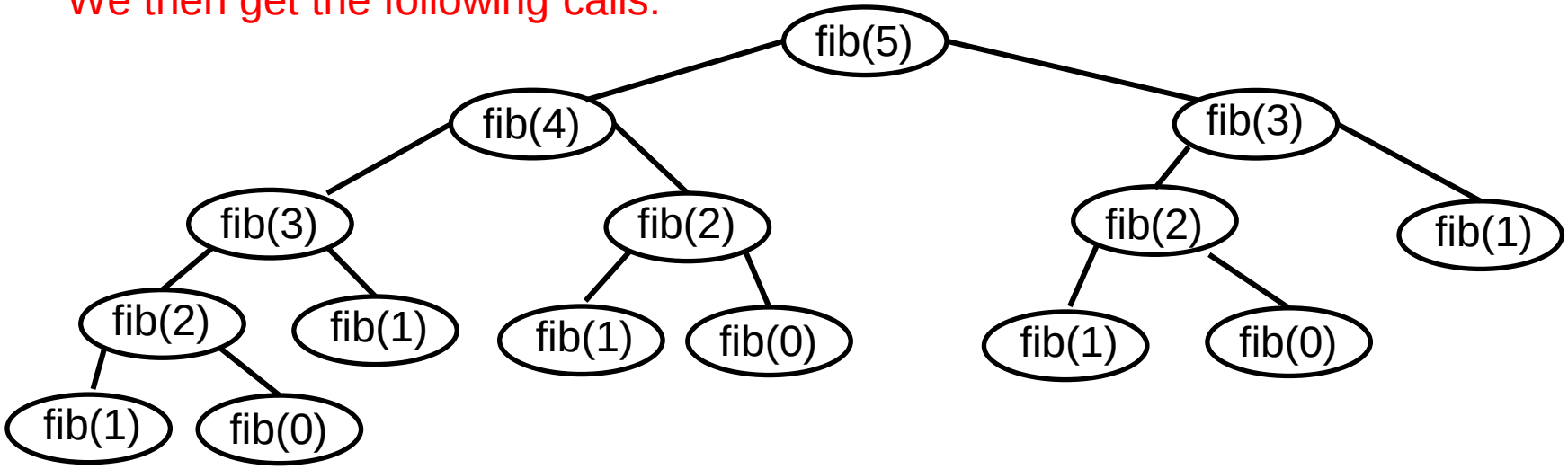
BUT: Who knew that this is *dynamic programming*?

The key formula: fib(n) = fib(n-1) + fib(n-2)

is then called the *Recurrence relation*

A less smart way to compute the fibonacci numbers

We could instead compute fib(n) by using the formula naively to recursive calls. We then get the following calls:



This gives a lot of recomputation (e.g. of *fib(2)* and *fib(3),* and this effect will only get worse as n gets higher.

But for more complicated cases it is not always easy to see that the iterative method that starts from low values is smarter, or even that it is possible!

NEXT SLIDE: Generilizing the fibonacci problem

Generalizing the fibonacci example

A more general case may have the following recurrence relation:

f(n) = < some function of f(0), f(1), …, f(n-1) >

To find the values og f(n) might be called:

«THE GENERAL ONE DIMENSIONAL DYNAMIC PROGRAMMING PROBLEM»

The function called «some function» above is, in Ch. 9 of the textbook, called *Combine*. Most of that chapter is concerned with optimazation, which is indeed often the case in DP, but we think more generally

The general problem above can be solved by storing the computed values in a one-dimentonal array, and compute the f-values from left to right by the actual recurrence relation:

0	1	2	3	4	5	. . .	n-1	n	…
f(0)	f(1)	f(2)	f(3)	f(4)	f(5)	. . .	f(n-1)	?	…

Note: We generally need an initialization that e.g. give values to f(0) and f(1).

Complexity: Assume that the *Combine* function is quite simple (e.g. as in Fibonacci):

Is this a polynomial time algorithm for computing f(n) for a given value n??

…….Thinking ….

It is in fact called a pseudo-polynomic function. This is often the case for DP-algorithms.

A simple two-dimenisional example

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

W:

12	5	35	7	11
4	29	8	19	14
8	3	19	46	1
37	84	78	55	62
26	13	40	33	12
21	60	27	18	17

P:

12	17	52	59	70
16				
24				
61				
87				
108				

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

NB: The shown red path is radomly chosen!

We use a new matrix P to store intermediate results: 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.

Questions (work for next week):

- How is the initialization made?
- Fill out the rest of the matrix according to the recurrence relation above? What orders can be used?
- How can we find the shortest path itself?
- What is the complexity of this algorithm?

Back to the string search problem:
We define the «edit distance» between two strings

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

T *P*

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

An idea for finding the Edit Distance

To solve this problem we will use an interger matrix D[0:m, 0:n] as shown below, where we imagine that the string P = P[1:m] is placed downwords along the left side of D, and T = T[1:n] is placed above D from left to right (at corresponding indices).

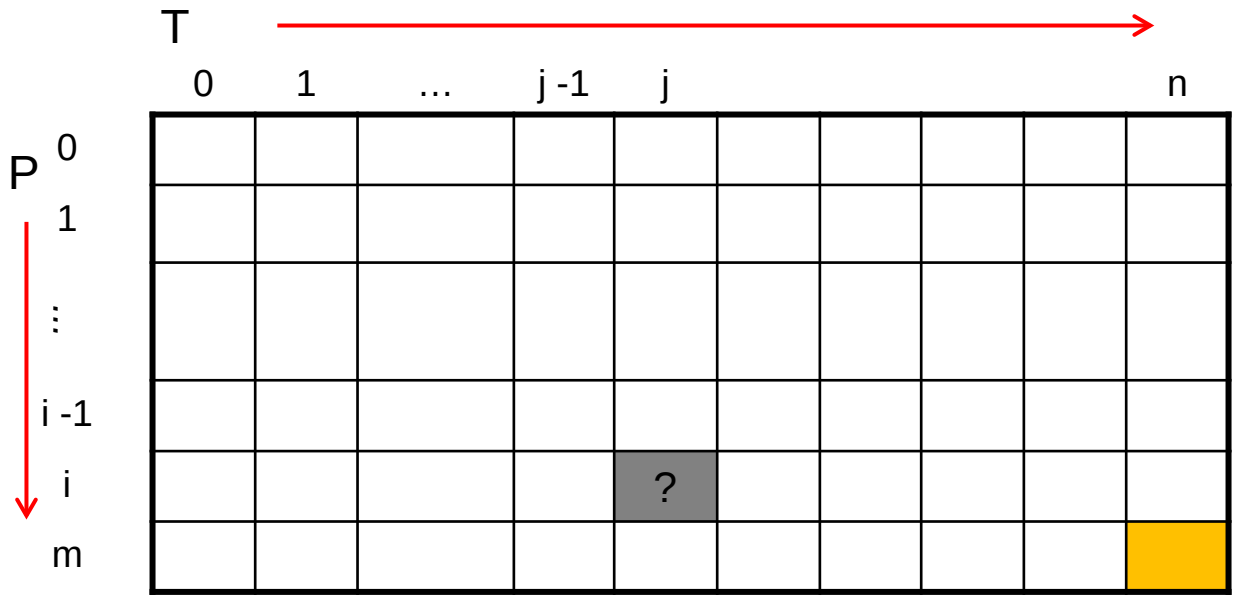
(This is a slightly different use of indices than in Sectons 20.5 and 9.4).

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

D[i, j] = Edit Distance between the strings P[1:i] and T[1:j]

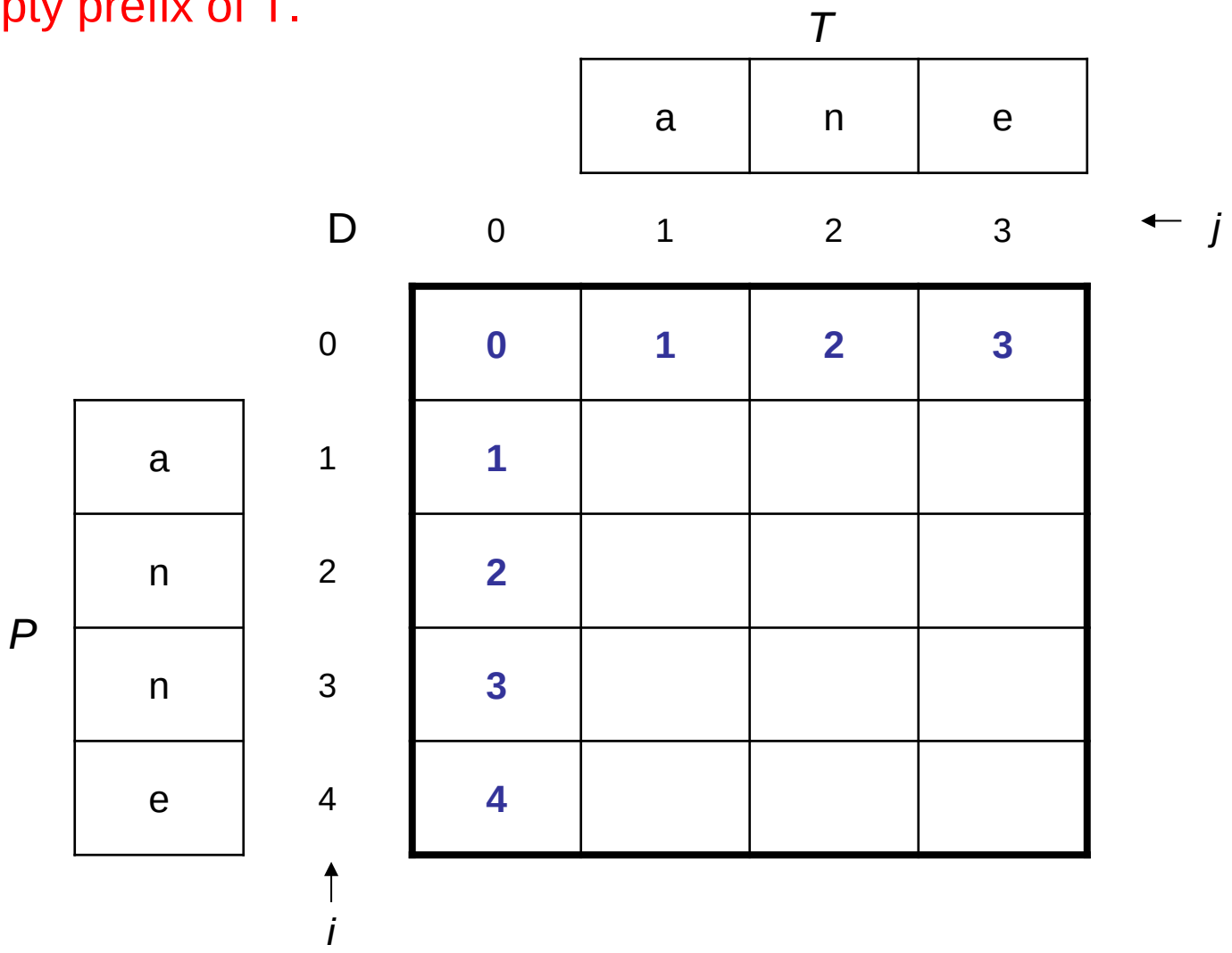
We will do this from «smaller» to «larger» index pairs (i, j), by taking column after column from left to right (but row after row from top to bottom would also work). The value we are looking for, ED(P, T), will then occur in D[m, n] when all entries are filled in.

The matrix *D*:



Example: P = «anne» and T = «ane»

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



More with P = «anne» and T = «ane»

We'll look a general cell D[i,j], and try to find how the value here can be computed from the values in the tree cells over and to the left.

We first, assume that Pi and Tj are the same letter (as below).
We know that P[1: i-1] can be transformed to T[1:j-1] in D[i-1, j-1] steps, and thus «T[1:j-1]'n'» can also be transformed into «D[i-1, j-1] 'n'» in D[i-1, j-1] steps.

		T	a	n	e
	D	0	1	2	3 ← *j*
P	0	**0**	**1**	**2**	**3**
a	1	**1**			
n	2	**2**	D[i-1, j-1]	. D[i-1, j]	.
n	3	**3**	D[i, j-1].	D[i, j]	.
e	4	**4**	. ↑	.	
	i				

Thus:
if P[i] and T[j] are the same letter then
D[i,j] = D[i-1, j-1]

More with P = «anne» and T = «ane»

We again look a general cell D[i,j], but we now assume that that P[i] and T[j] are NOT the same letter.

We know that:

P[1: i-1] can be transformed to T[1: j-1] in D[i-1, j-1] steps, and we can thus transform «T[1: j-1]‘x’» to «P[1: j-1]‘y’» in D[i-1, j-1] +1 steps.

		T	a	x	e
	D	0	1	2	3 ← *j*
P	0	**0**	**1**	**2**	**3**
a	1	**1**			
n	2	**2**	D[i-1, j-1]	. D[i-1, j]	.
y	3	**3**	D[i, j-1].	D[i, j]	.
e	4	**4**	. ↑	.	
	i				

Likewise we get:

We can transform «T[1: j-1]» into «P[1: i-1]‘y’» in D[i, j-1] +1 steps.

«T[1: j-1]» into «P[1: i-1] ‘y’» in D[i-1, j] +1 steps.

Thus we can do the transformation from «T[1: j-1]‘x’» into «P[1: i-1] ‘y’» in the minimum number of steps used in these three scenarios. We therefore obtain the formula on next page.

The general recurrence relation becomes

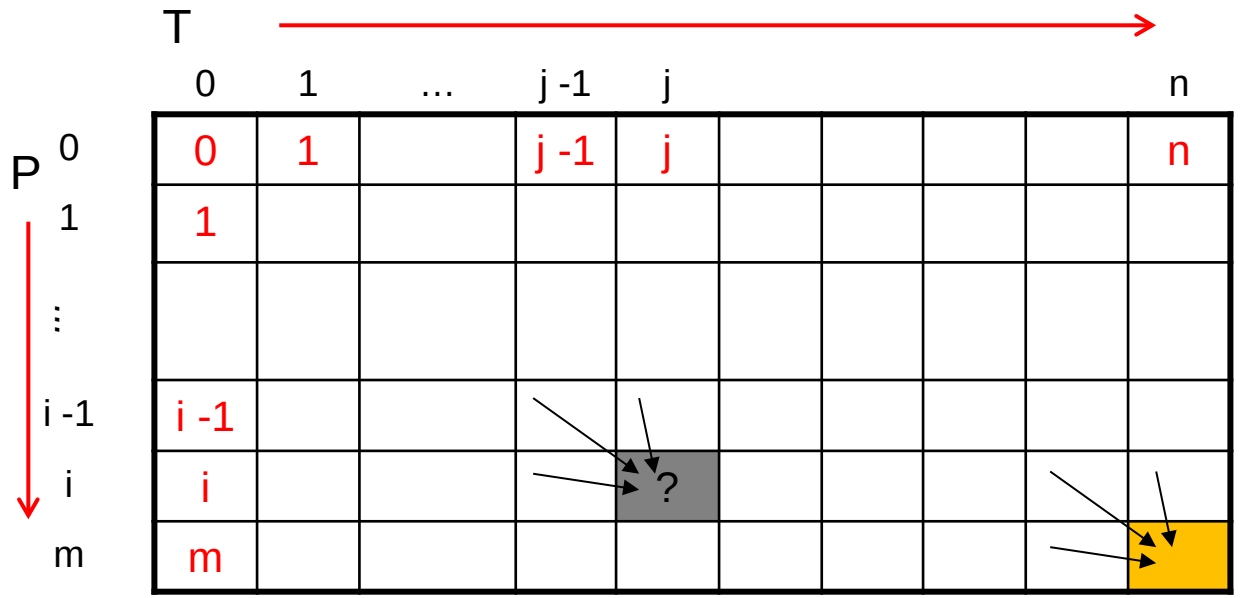
- To fill in this matrix D we in fact used the relation indicated below.
- Note that the value of D[i,j] only depends on entries in D with «smaller» index-pairs: D[i-1,j-1], D[i-1,j], and D[i,j-1]

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

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

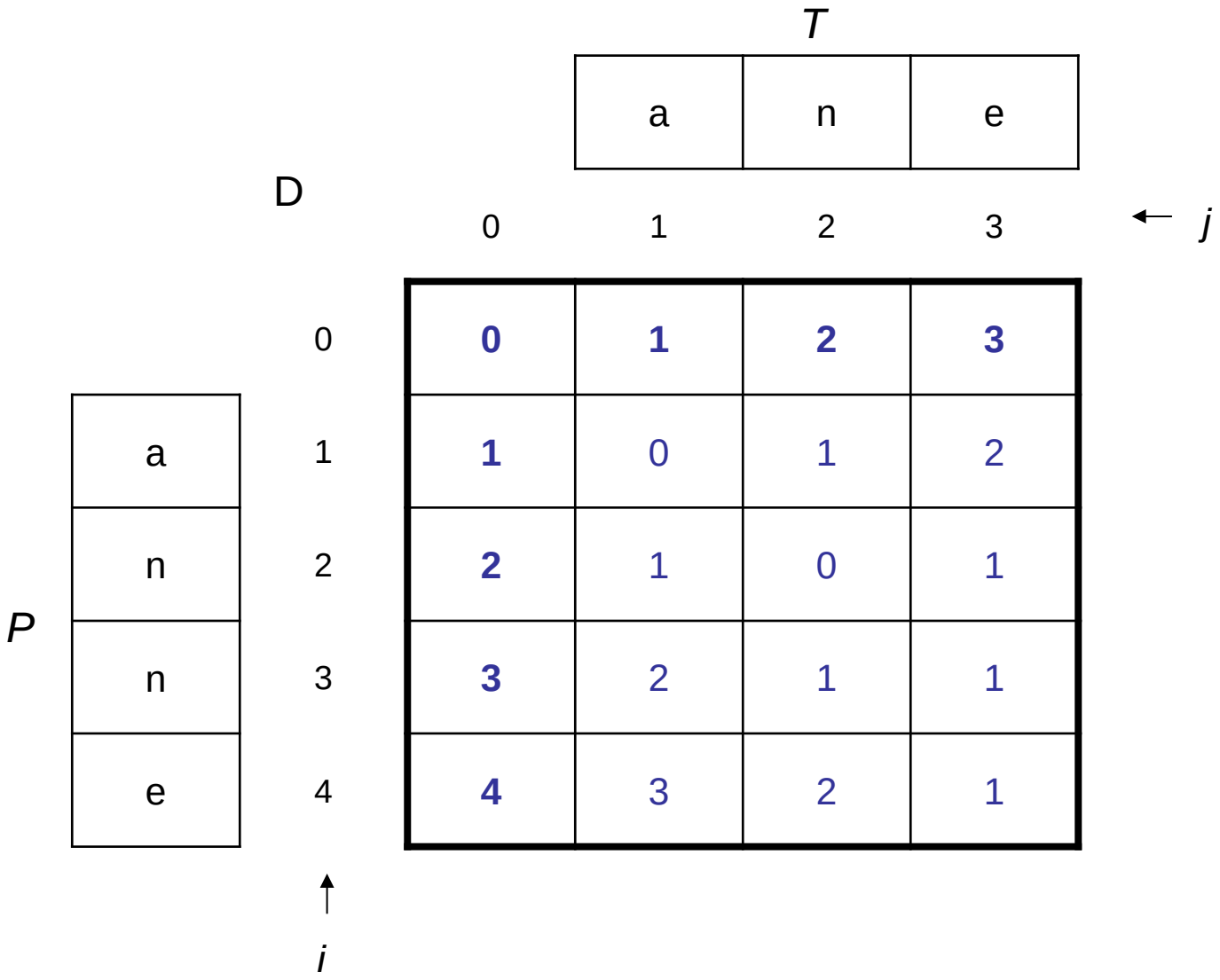
The equalities at the last line can be used to initialize the matrix (shown in red).

The matrix *D*:



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)



A program for computing the edit distance

```
function EditDistance ( P [1:m ], T [1:n ] )
    for i ← 0 to m do D[ i, 0 ] ← i   endfor      // Initialize row zero
    for j ← 1 to n do D[ 0, j ] ← j  endfor       // Initialize column zero

    for i ← 1 to m do
        for j ← 1 to n do
            If P [ i ] = T [ j ] then
                D[ i, j ] ← D[ i -1, j - 1 ]
            else
                D[ i, j ] ← 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):

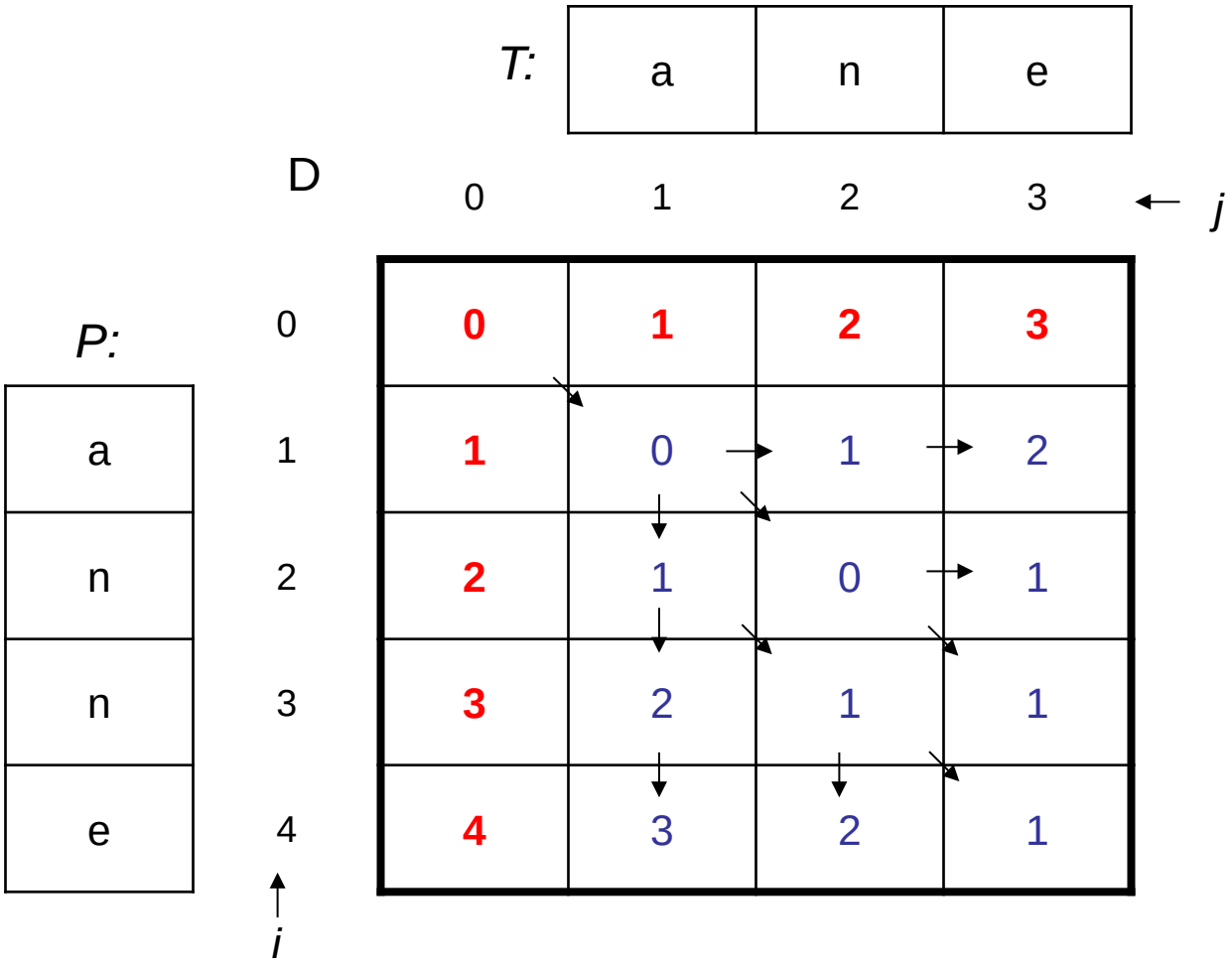
(1,1) (1,2) … (1,n)
(2,1) (2,2) … (2,n)
…
(m,1) … (m,n)

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 always computed before D[i, j]

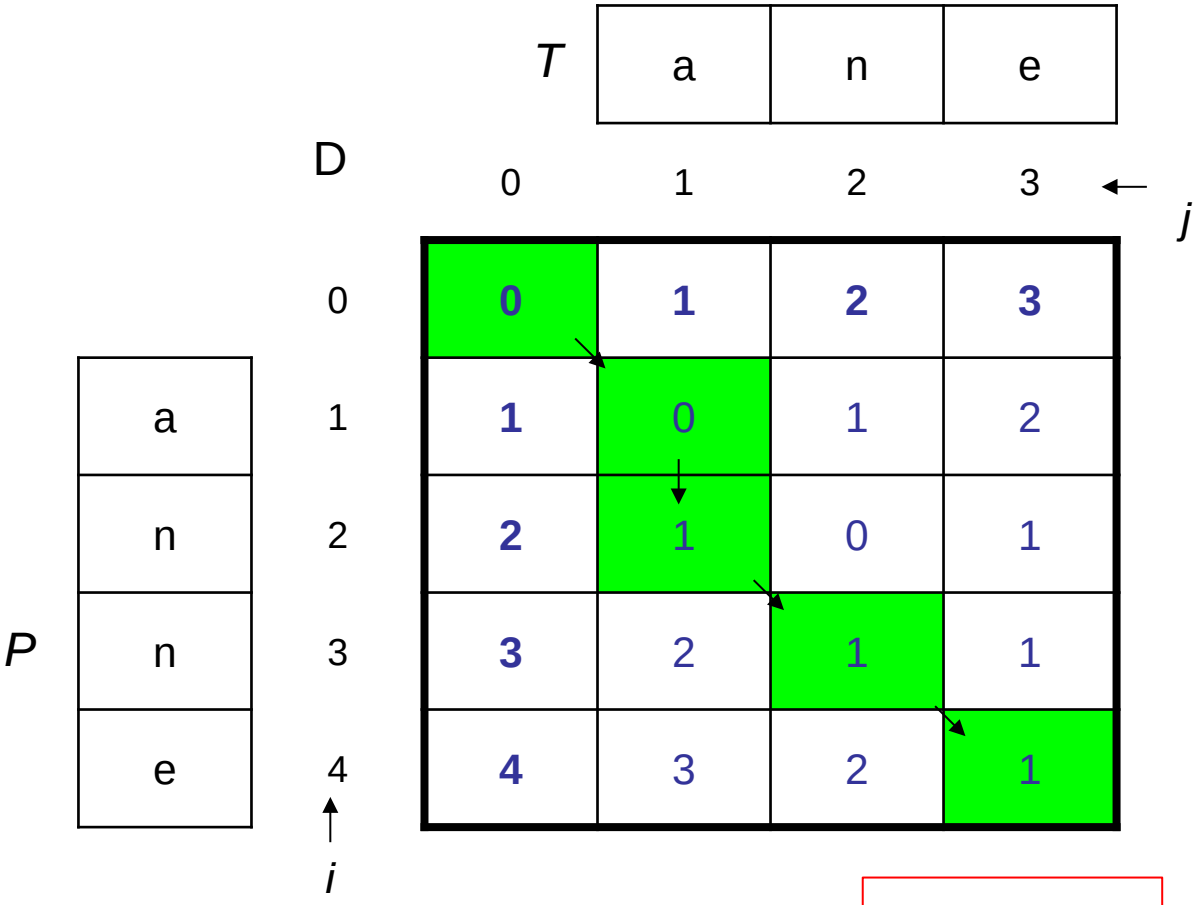
Our old example: Finding the edit steps

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



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



The result can be visualized as follows:

a n n e
a . n e

Diagonally, and P[i] = T[j]:
No edit needed. Occur e.g.for D[3, 2].

Diagonally, and P[i] ≠ T[j]:
- Substitution Occur e.g. for D[3, 3] (*not* used in the shortest edit path)

Downwards (and thus P[i] ≠ T[j]):
- A letter is deleted from P
Occur e.g.for D[2, 1]

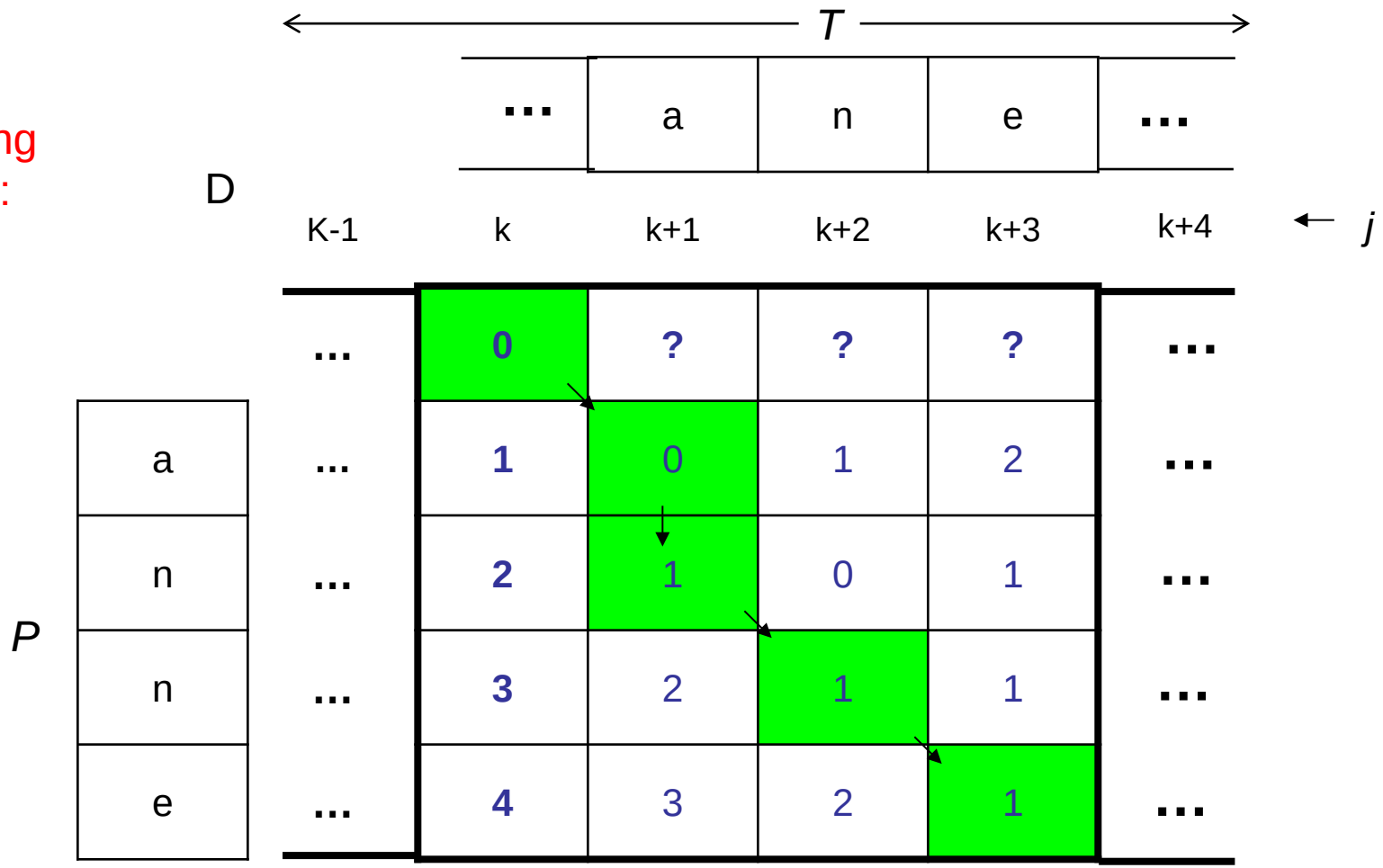
Towards the right (and thus P[i] ≠ T[j]):
A letter is added to P
Occur e.g. for D[1, 2] (but is *not* used in the shortest edit path)

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 problem will be an exercise later this week!

Something like this?:



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:
u t t x v

The human genome:
b s u t t v r t o f i g u t t v x l 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 later this semester.

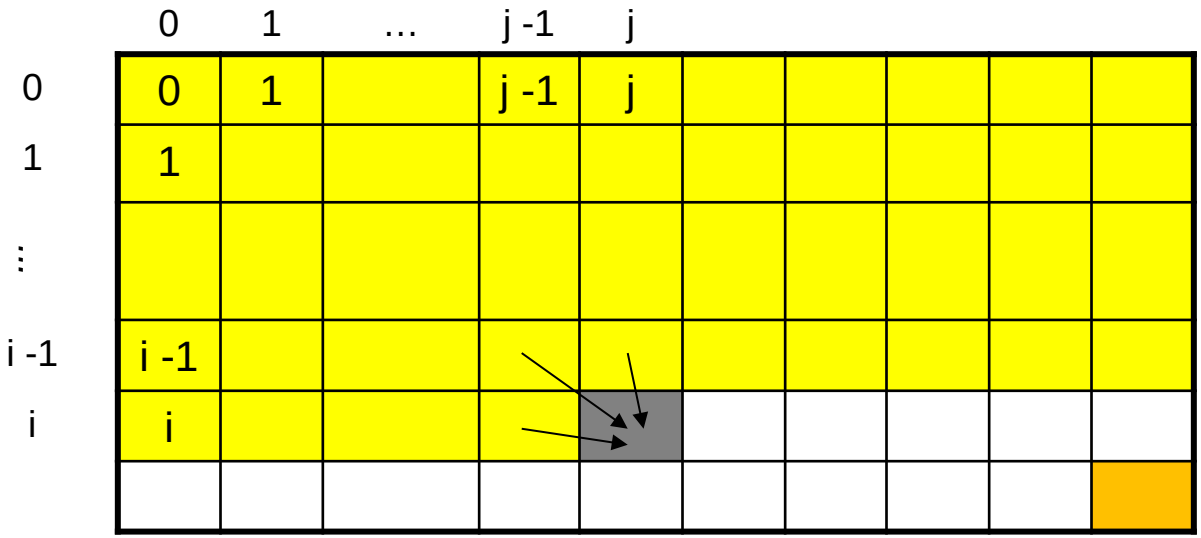
About Dynamic Programming in general

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

:

In our example:

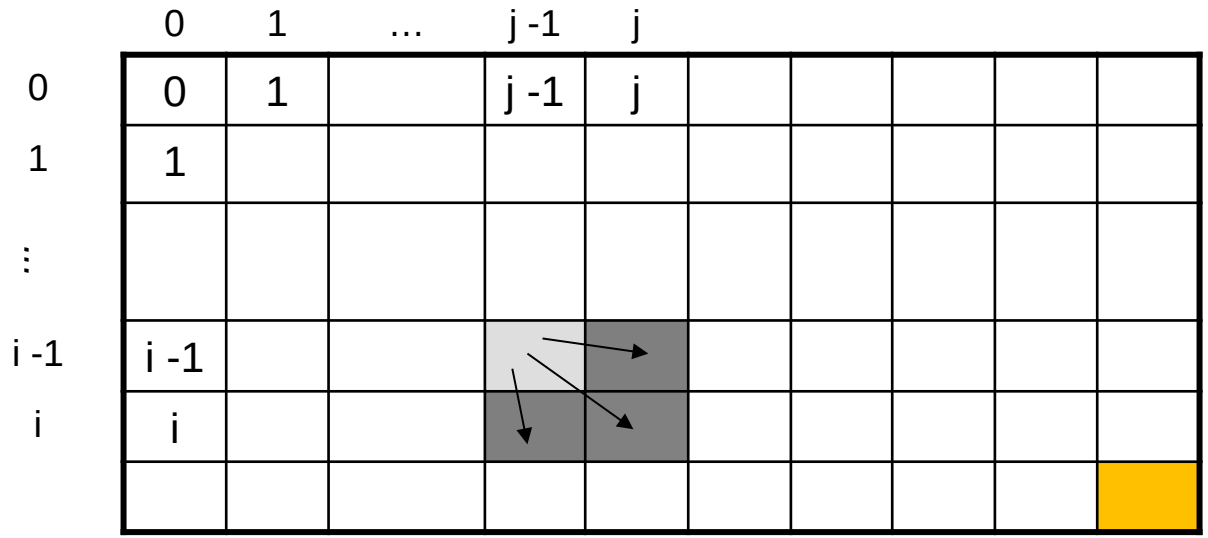
'The values of the yellow area is computed when the white value is to be computed



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.



A rather formal basis for Dynamic Programming

You might be better equipped for the exam if you have been trough it!

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

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 $\{ J_1, \ldots, J_k \}$ called the *base of I*, written $B(I)=\{ J_1, J_2, \ldots, J_k \}$ (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

  $$s(I) = Combine(I, s(J_1), s(J_2), \ldots, s(J_k))$$

  This is called the «recurrence relation» of the problem.
- For an instance I, we can set up a sequence of instances $< L_0, L_1, \ldots, L_m >$ with growing sizes, and where L_m is the problem we want to solve, and so that for all $p \leq m$, all instances in $B(L_p)$ occur in the sequence *before* L_p.
- The solutions of the instances $L_0, L_1, \ldots, L_m$ 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.

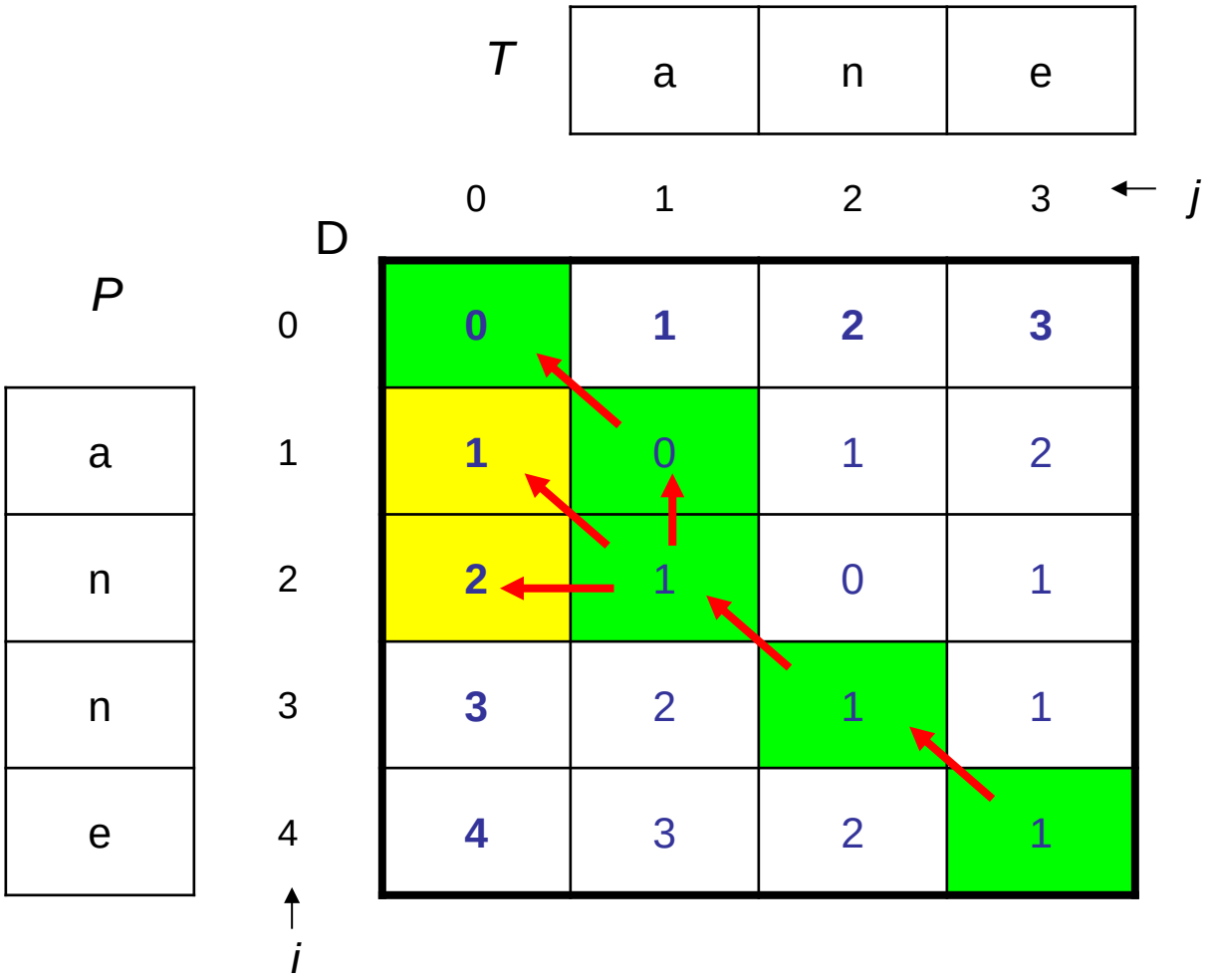
2. «Top-Down» Dynamic Programming

A drawback with traditional dynamic programming is that one usually solves a number of smaller instances that turn out not to be needeed 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 in the table whether it is already solved, and if so we only use the stored solution. Otherwise we do recursive calls, and stores 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.

Another example: Optimal Matrix Multiplication

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

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 $i = 0, 1, \ldots, n-2$

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):

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

The cost (the number of simple (scalar) multiplications) for 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, slide 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 showing that the muitiplication order has significans:

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, slide 3

Given a sequence of matrices $M_0, M_1, \ldots, M_{n-1}$. 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 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 parenthesizaton of $M_0, M_1, \ldots, M_{n-1}$, 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 a lot of instances of smaller sizes.

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

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

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

Optimal matrix multiplication 4

Let $d_0, d_1, \ldots, d_n$ be the dimensiones of the matrices $M_0, M_1, \ldots, M_{n-1}$, 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 parenthesizaton of $M_i, M_{i+1}, \ldots, M_j$.

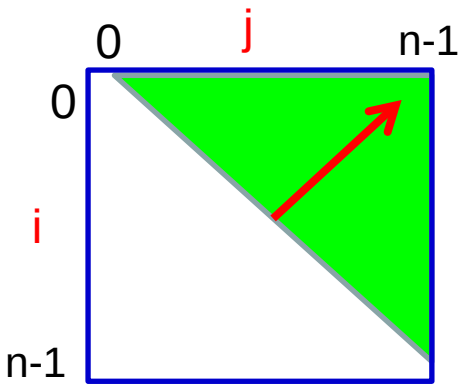
Thus the value we are interested in is $m_{0,n-1}$

The recurrence relation for $m_{i,j}$ will be:

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

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

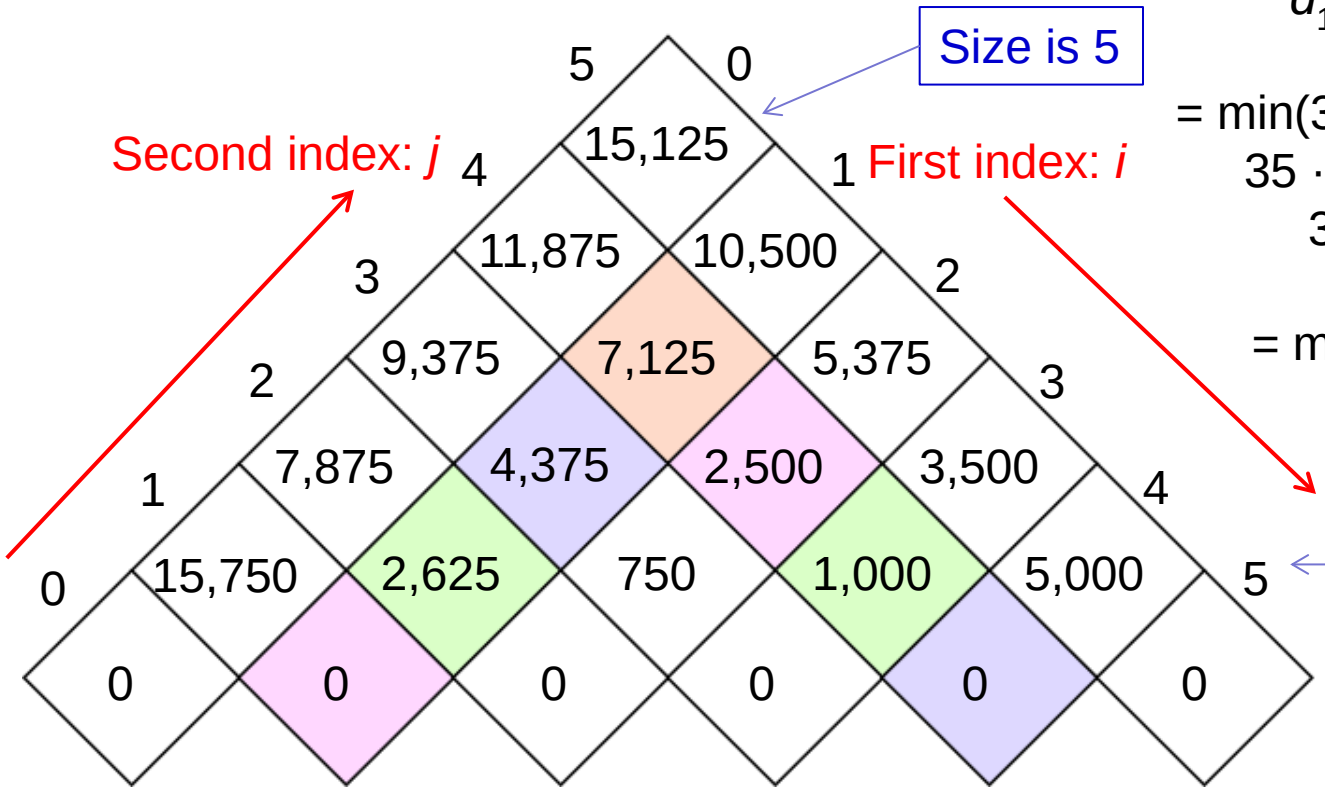
Here, importantly, the values $m_{k,l}$ that we need for computing $m_{i,j}$ are all for *smaller* instances. With usual indexing this means we shall fill the green area from the diagonal towards the upper right corner, as shown by the red arrow. In the next slide this green triangle is turned 45 degrees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



The table: Optimal matrix multiplication

d	30	35	15	5	10	20	25

The values $m_{i,j}$:



Example:

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

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

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

$= 7125$

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

Program: Optimal matrix multiplication

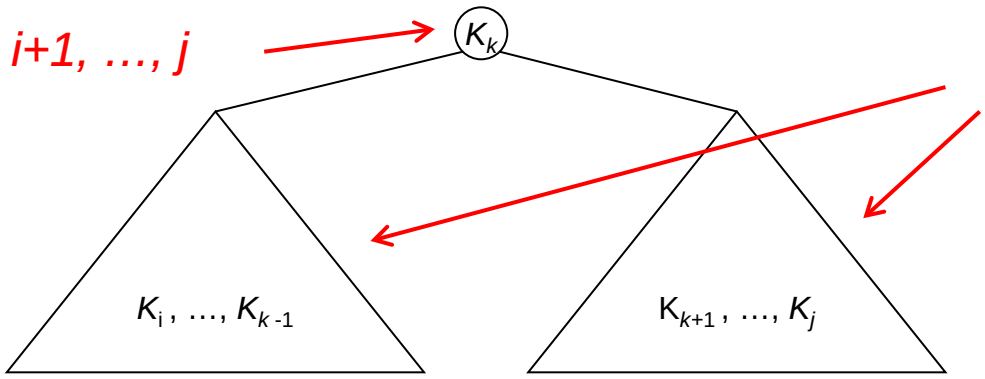
```
function OptimalParens( d[0 : n – 1] )
    for i ← 0 to n-1 do
        m[i, i] ← 0
    for diag ← 1 to n – 1 do
        for i ← 0 to n – 1 – diag do
            j ← i + diag
            m[i, j] ← ∞   // Relative to the scalar values that can occur
            for k ← i to j – 1 do
              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

Optimal search trees

(Not part of the curriculum!)

- 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 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 (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.

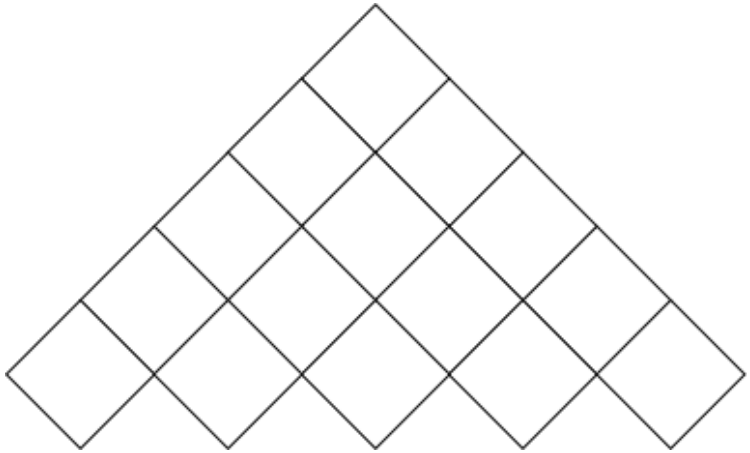
Try with *k= i, i+1, …, j*

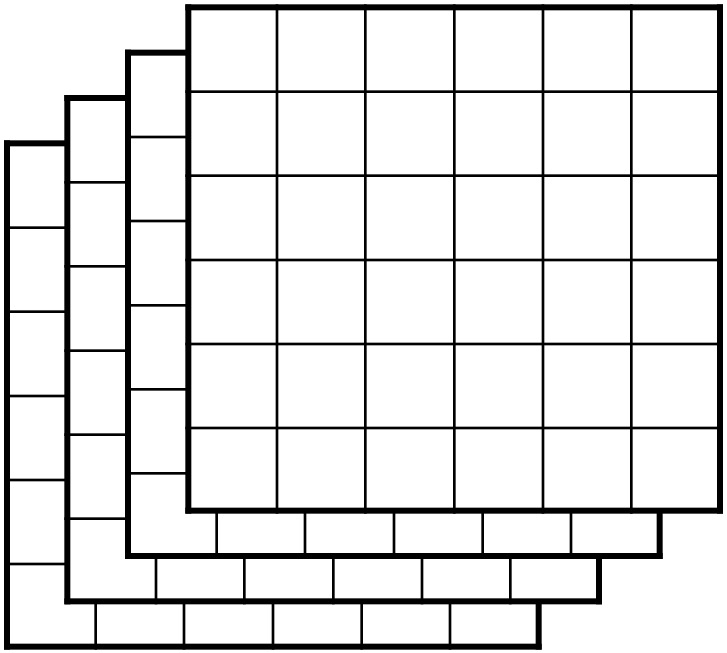


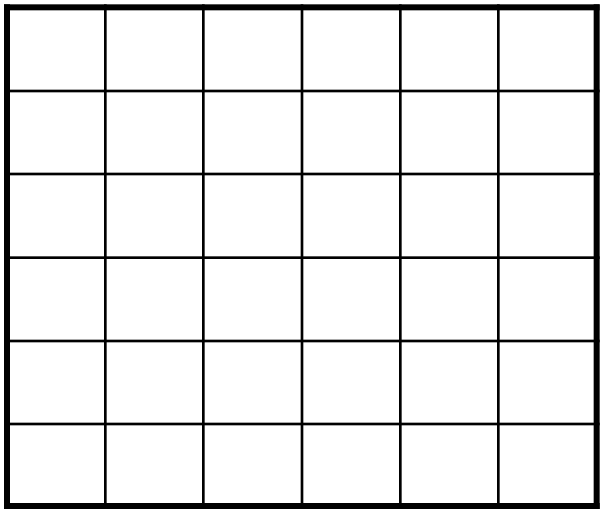
The optimal values and form for these subtrees are already computed, when we here try with different values K_k at the root

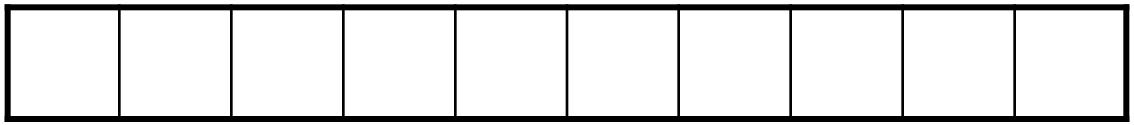
Dynamic programming in general:

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).