INF4130: Dynamic Programming

September 2, 2014

DRAFT version

- In the textbook: Ch. 9, and Section 20.5
- Chapter 9 can also be found at the home page for INF4130
- These slides were originally made by Petter Kristiansen, but are adjusted by Stein Krogdahl.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:
- I (SK) think the one used here is easier to understand (but that might indeed be a matter of taste!):

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.

We start with an example:

Searching for *similar* strings (Ch. 20.5)

This is relevant e.g. for research in genetics

A string P is a <u>*k*-approximation</u> 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 <u>Edit Distance</u>, $ED(P,T)$, between two strings T and P is the smallest number of such operations needed to convert T to P *(or P to T!).*

Example.

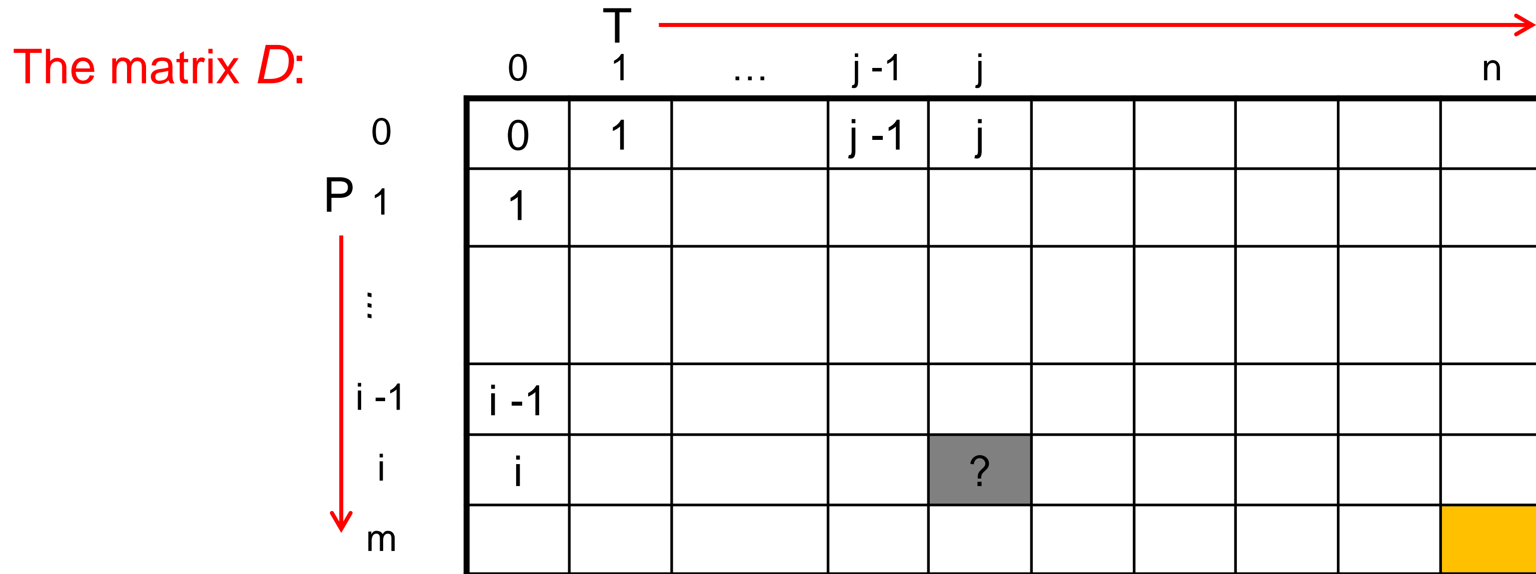
logarithm → alogarithm → algarithm → algorithm (Steps: +a, -o, a->o)

T ... *P*

Thus ED(”logarithm”, ”algorithm”) = 3

Finding the edit distance

- Given two strings *T* and *P* of length m and n respectively.
- We want to find *ED*(*P, T*).
- We will use a two-dimentional matrix *D*, and we hope to fill it in so that:
  $D[i, j] = ED(P[1:i], T[1:j])$. The size of this instance is $i + j$
- We imagine that the string *P* is written downwards along the left edge, and that *T* is written from right to left above the matrix:



- 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 should these be filled as indicated above?

Finding the edit distance

See discussion of a similar problem in Ch. 9.4

From previous slide: Given two strings T and P of length m and n respectively.

- We want to find $ED(P, T)$.
- We will use a two dimentional matrix D, and hope to fill it so that:

$$D[i, j] = ED(P[1:i], T[1:j]).$$

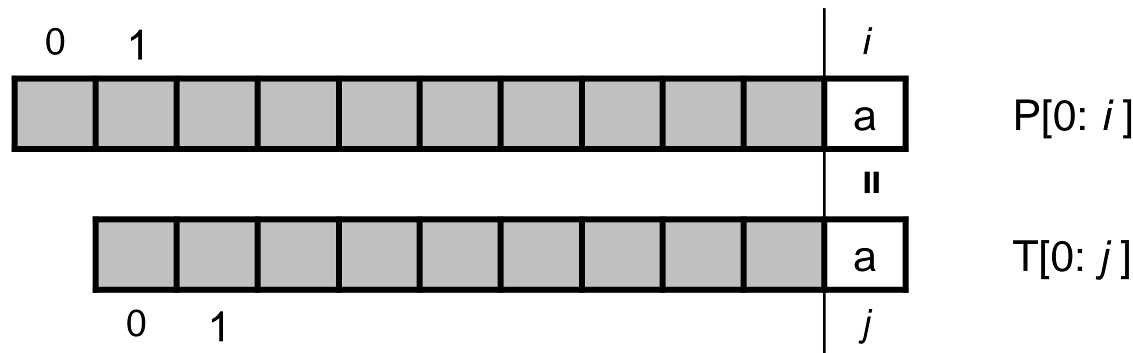
- Thus, the answer for the full strings P and T will occur in D[m,n]

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

$$D[i-1, j-1],\ D[i, j-1], \text{ and } D[i-1, j]$$

which all have smaller sizes than D[i, j]. There are two cases:

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

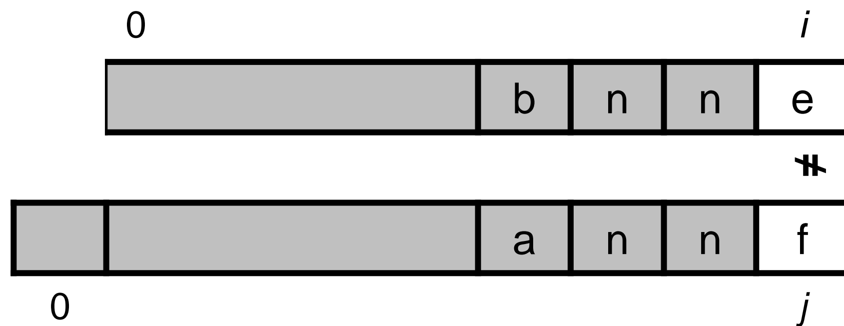


Why is it not possible to obtain a better ED?

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

We choose the best of the following three possibilities (but why is this enough?):

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

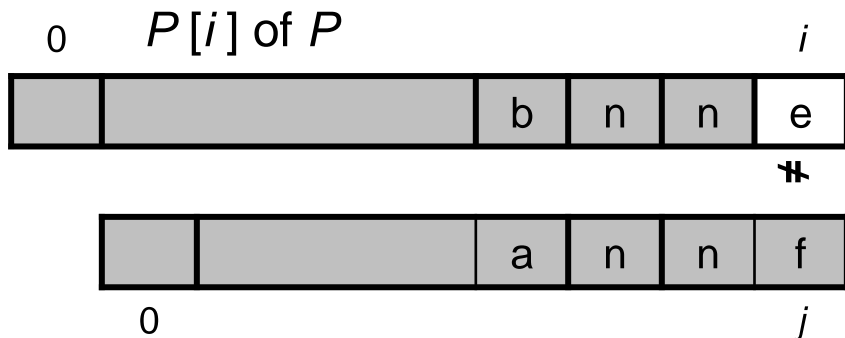


P[1: *i*]

T[1: *j*]

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 – corresponds to removing the last symbol $P[i]$ of P

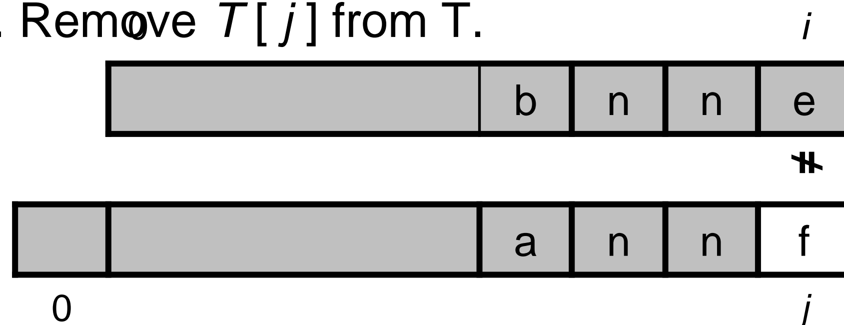


P[1: *i*]

T[1: *j*]

ED[*i*, *j*] would be $D[i-1, j] + 1$ (*ED* between the gray areas plus 1)

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



P[1: *i*]

T[1: *j*]

ED[*i*, *j*] would be $D[i, j-1] + 1$ (*ED* between the gray areas plus 1).

Computing edit distance

Thus, a recursive expression for $D[\, i, j\,]$ is:

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

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

	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 size 0.

Computing edit distance

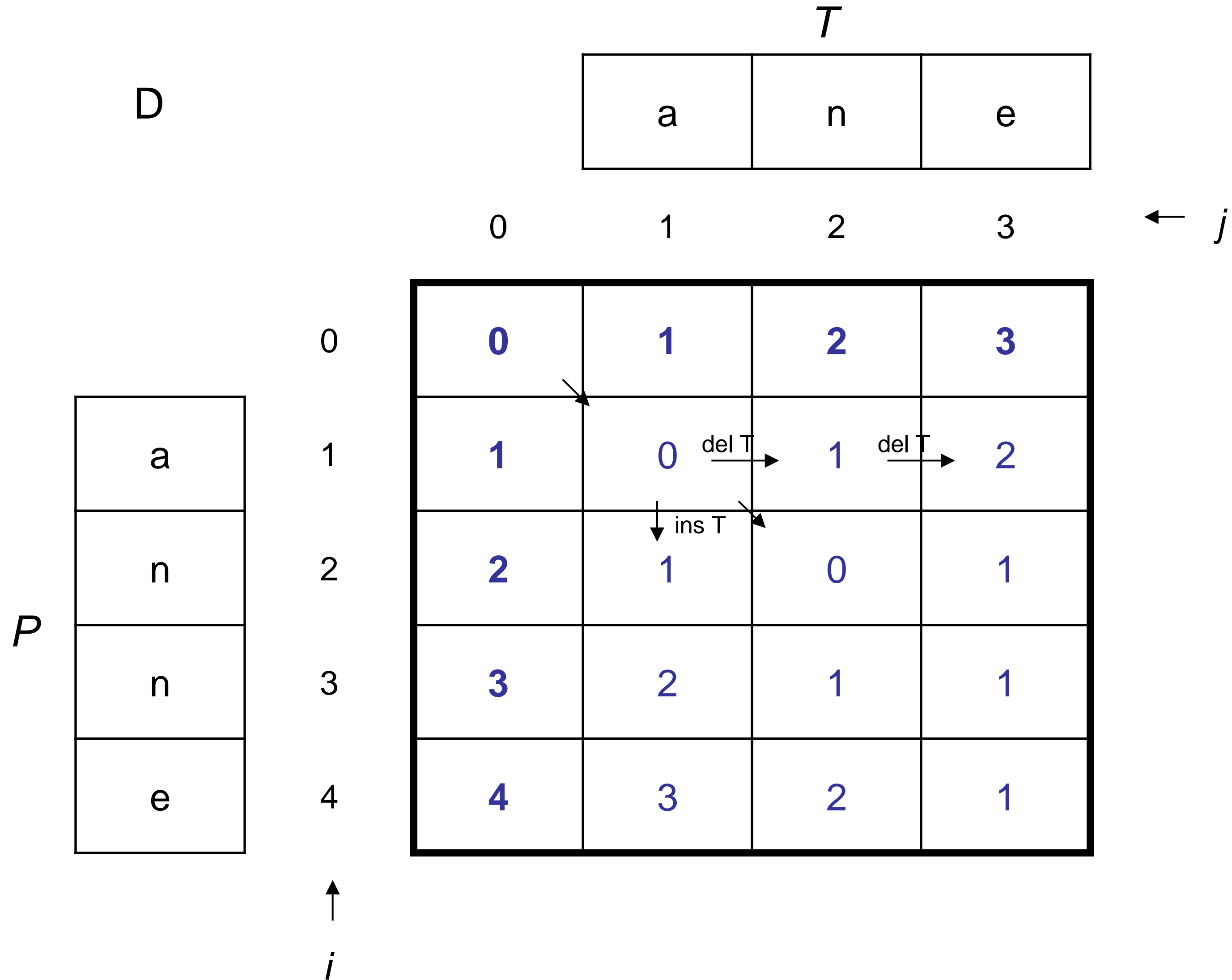
```
function EditDistance ( P [1:n ], T [1:m ] )
    i ← 0
    j ← 0
    for i ← 0 to n do D[ i, 0 ] ← i
    for j ← 1 to m do D[ 0, j ] ← j
    for i ← 1 to n do
        for j ← 1 to m do
            If P [ i ] = T [ j ] then
                D[ i, j ] ← D[ i -1, j - 1 ]
            else
                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[ 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)

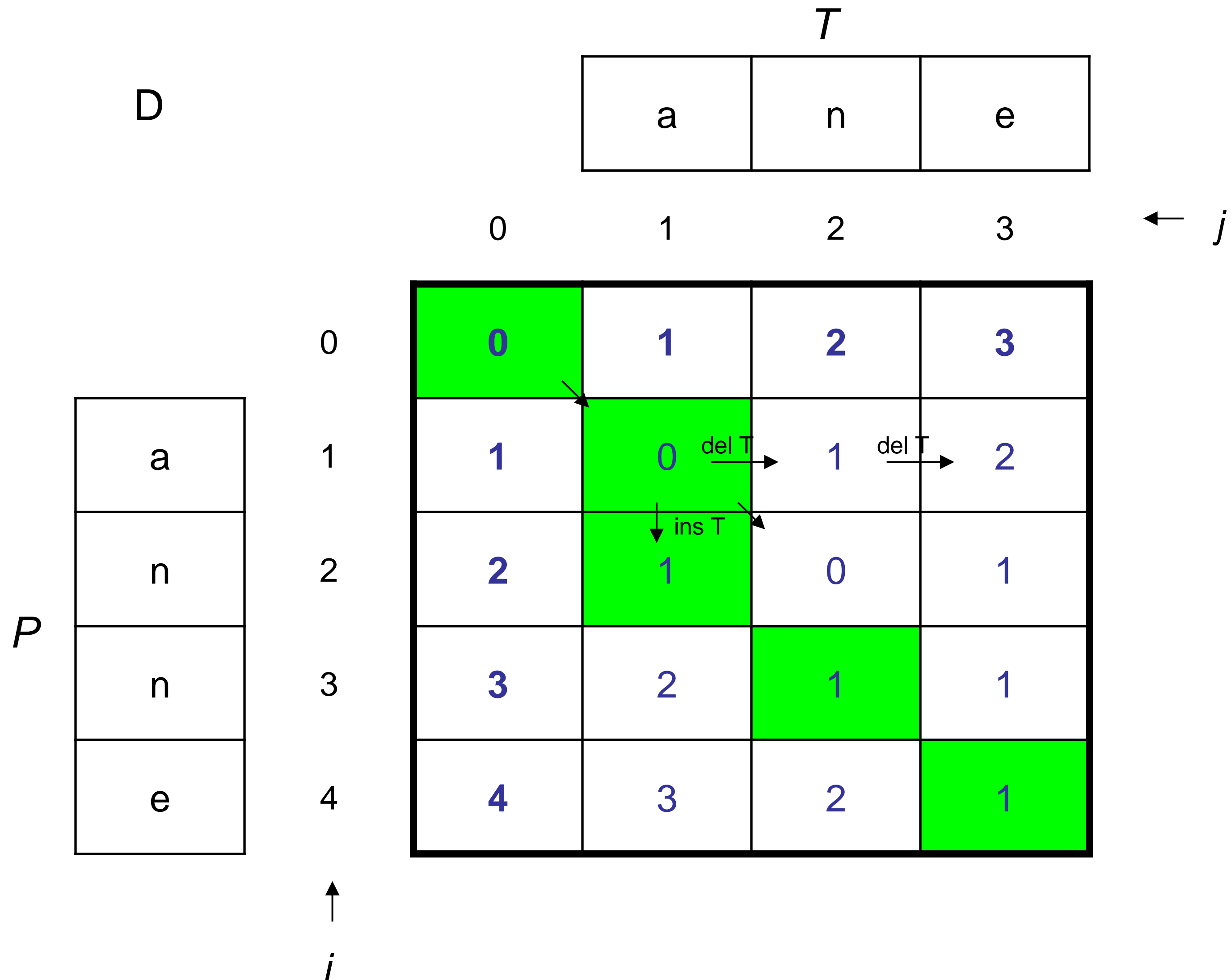
This is OK as also this order ensures that the smaller instances are solved before they are needed to solve a larger instance. An order strictly following increasing size would also work fine, but is slightly more complex to program (following diagonals).

Example



Computing edit distance

Example



Generally about 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 A of size *n* there is a set of instances B_1, B_2, … ,B_m, all with sizes less that n, so that we can find an (optimal) solution of A if we know the (optimal) solution of the B_i-problems.

Example:

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

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

Generally about dynamic programming - 2

- In the textbook (page 265) the solution to the instance A is called S, and that of each B_i is called S_i. The way to find S from the S_i-s is written:

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

For our example problem:

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

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

Generally about dynamic programming - 3

- 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?

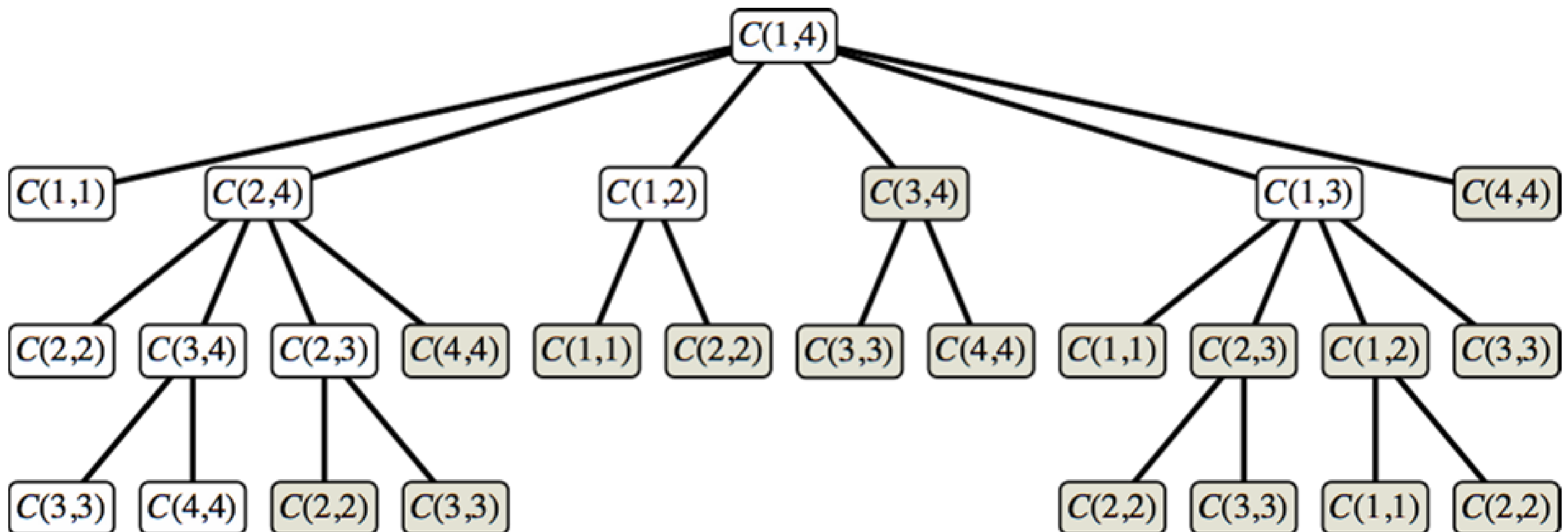
- Thus, if we compute and store (in a table of a suitable format), the solutions to all relevant instances of a given size before looking at instances of larger sizes, we will always know the arguments to the Combine-function when we need them for computing the solution of an instance of larger size.

- 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

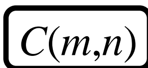
- As indicated on the previous slide, Dynamic Programming is useful if the solution to a certain instance is used in the solution of many instances of larger size.
- In the problem C below, an instance is given by some data (e.g two strings) and by two intergers *i* and *j.* The corresponding instance 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, and that all terminal nodes have size 0.



Number of solutions needed

- If the number of solutions to different smaller instances that are needed to find the soluton to a certain instance may very big (e.g. exponential in the size of the instance), then the resulting algoritm will usually not be practical.

- Instead, this number should at least 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)

Dynamic Programming (bottom up)

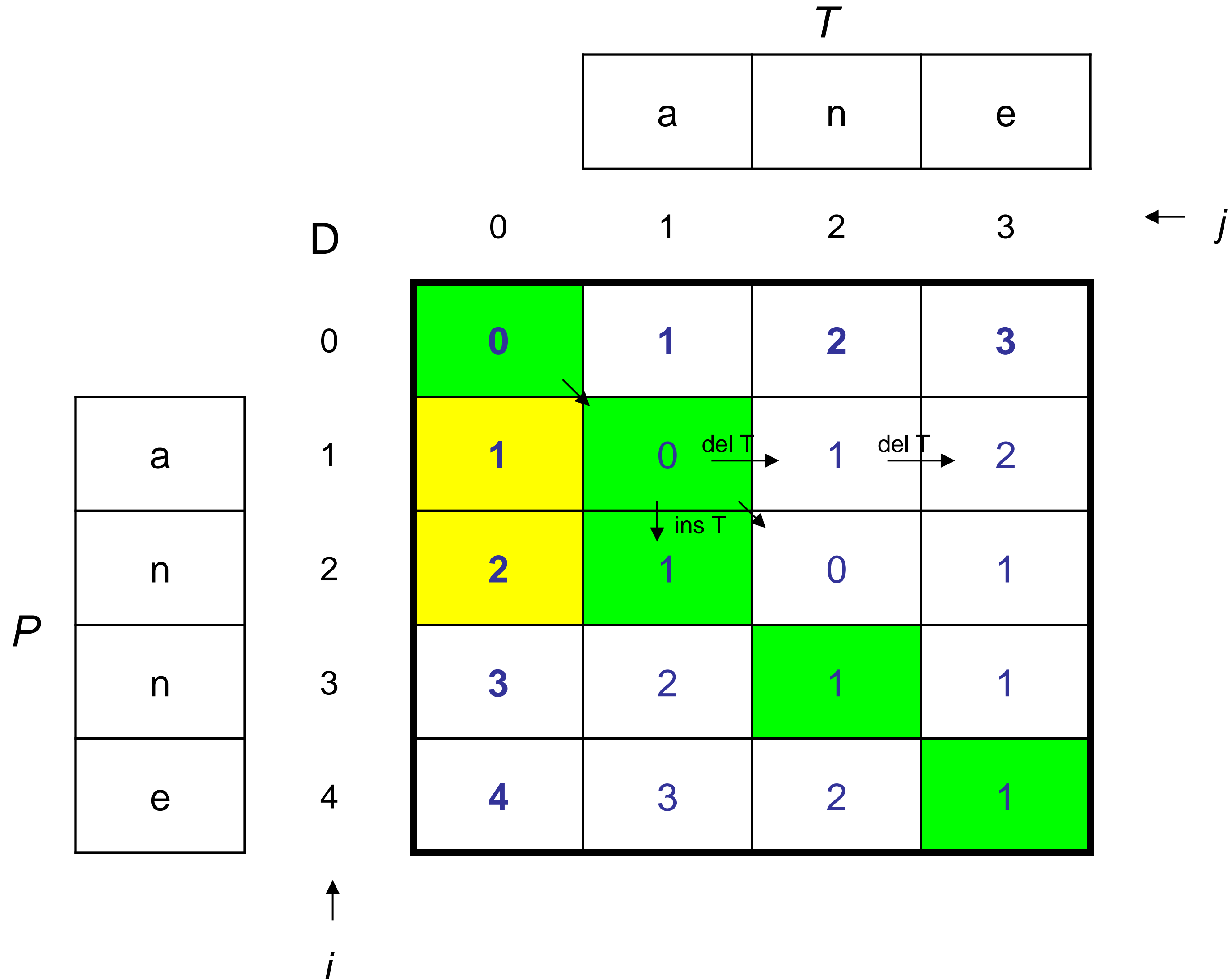
- Is traditionally performed bottom-up. All relevant smaller instances are solved first, and the solutions are stored in a table.
- Works best when the answers to smaller instances are needed by many larger instances.

«Top-Down» dynamic programming (called Memoization**)**

- 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 solutions into the same table as above (see later slides).
- The table entries then need a special marker «not computed», which should be the initial value of the entries.

«Top-Down» dynamic programming
”Memoization”

You only have to compute the colored entries below



New 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 · 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 vary a lot between the differnt alternatives. We want to find the 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 parameterization with 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 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-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}, …, M_j.

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

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

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

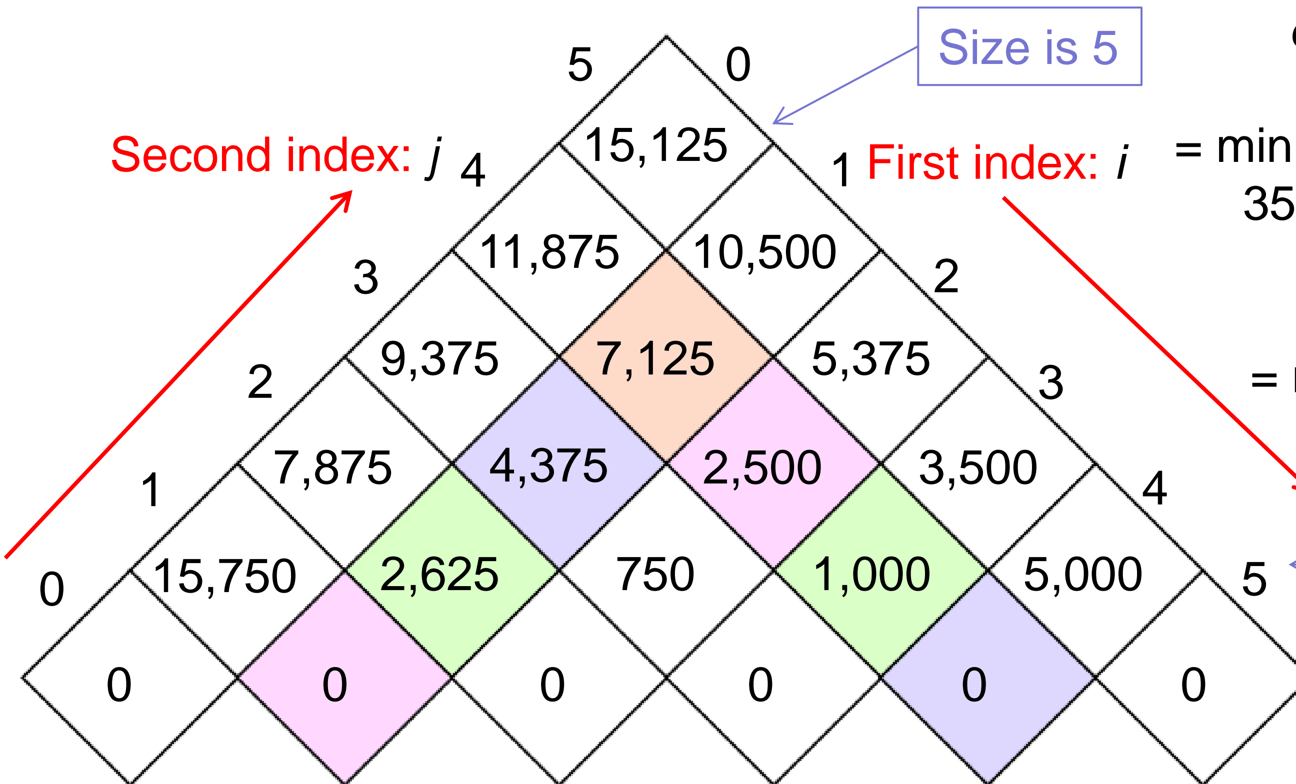
$$m_{i,i} = 0 \text{, for all } 0 \le i \le n-1$$

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

The values $m_{i,j}$:



Size is 1

Size is 0

Example

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

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

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

$$= 7125$$

Optimal matrix multiplication

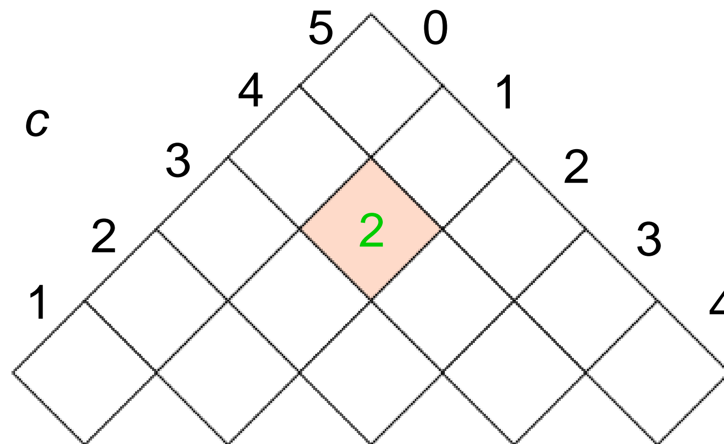
Remembering the best partitions

$M_1 \cdot M_2 \cdot M_3 \cdot M_4$

$= M_1 \cdot (M_2 \cdot M_3 \cdot M_4)$

$= (M_1 \cdot M_2) \cdot (M_3 \cdot M_4)$

$= (M_1 \cdot M_2 \cdot M_3) \cdot M_4$



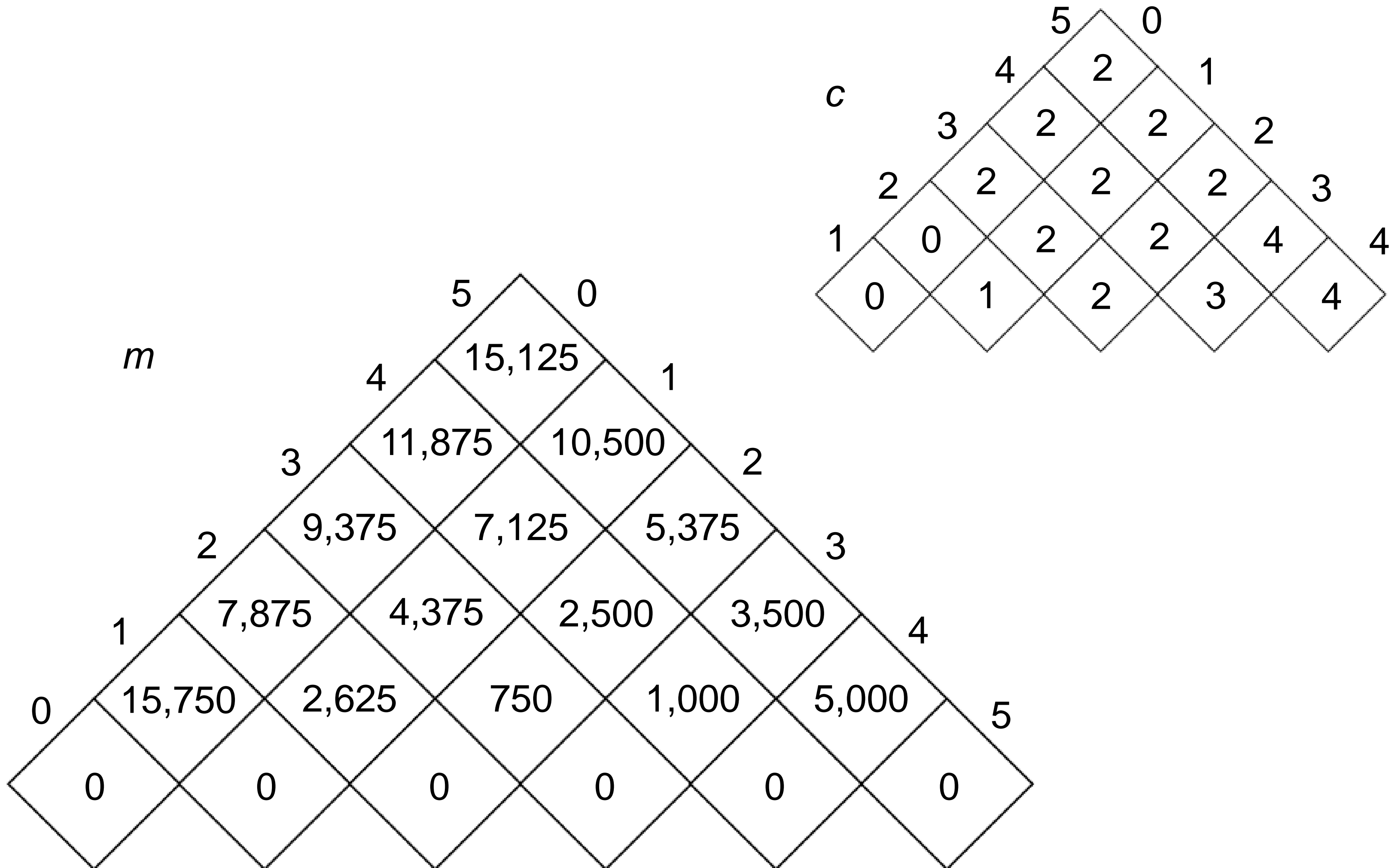
m

5 0
15,125
4 1
11,875 10,500
3 2
9,375 7,125 5,375
2 3
7,875 4,375 2,500 3,500
1 4
15,750 2,625 750 1,000 5,000
0 5
0 0 0 0 0 0

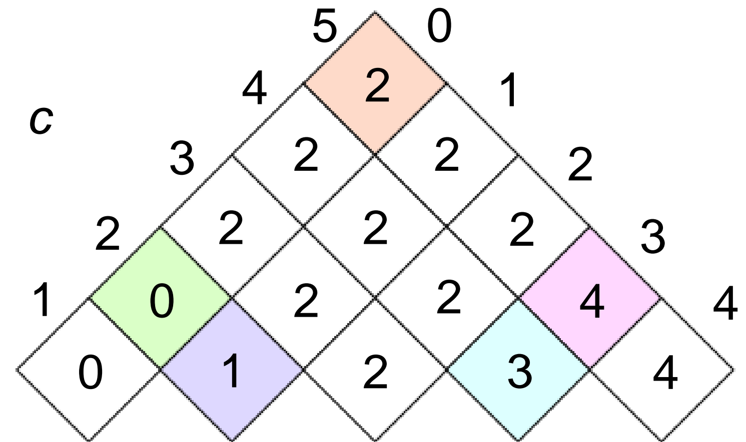
Tabel c: Remembering the optimal partition points. The optimal partition point of $M_1 \cdot M_2 \cdot M_3 \cdot M_4$ is after M_2

Optimal matrix multiplication

Remembering the best partitions



Optimal matrix multiplication



The optimal parenthesizaton is thus:

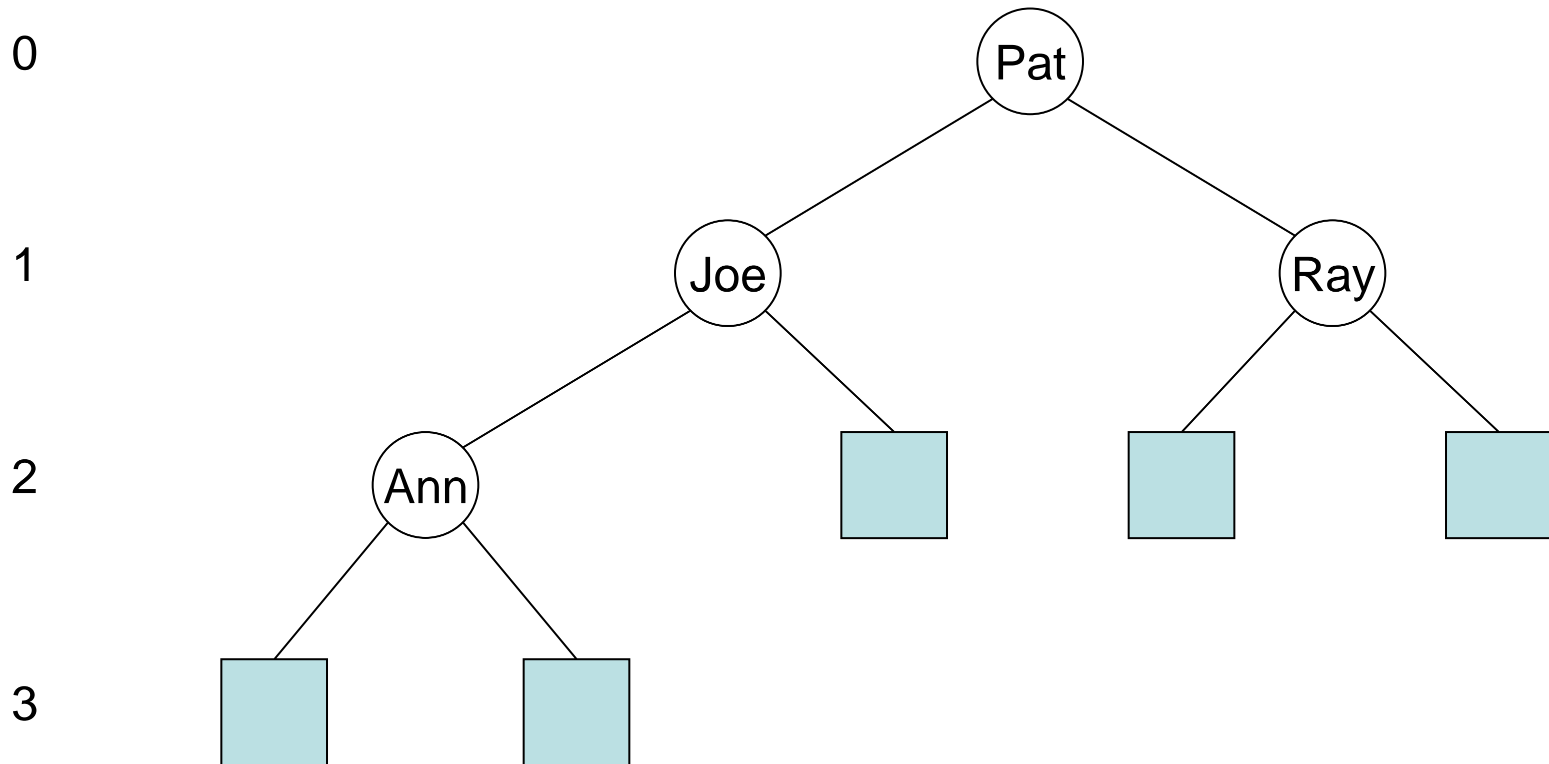
$M_0 \cdot M_1 \cdot M_2 \cdot M_3 \cdot M_4 \cdot M_5$
$= (M_0 \cdot M_1 \cdot M_2) \cdot (M_3 \cdot M_4 \cdot M_5)$
$= ((M_0) \cdot (M_1 \cdot M_2)) \cdot ((M_3 \cdot M_4) \cdot (M_5))$

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 2013, but maybe for 2014?)



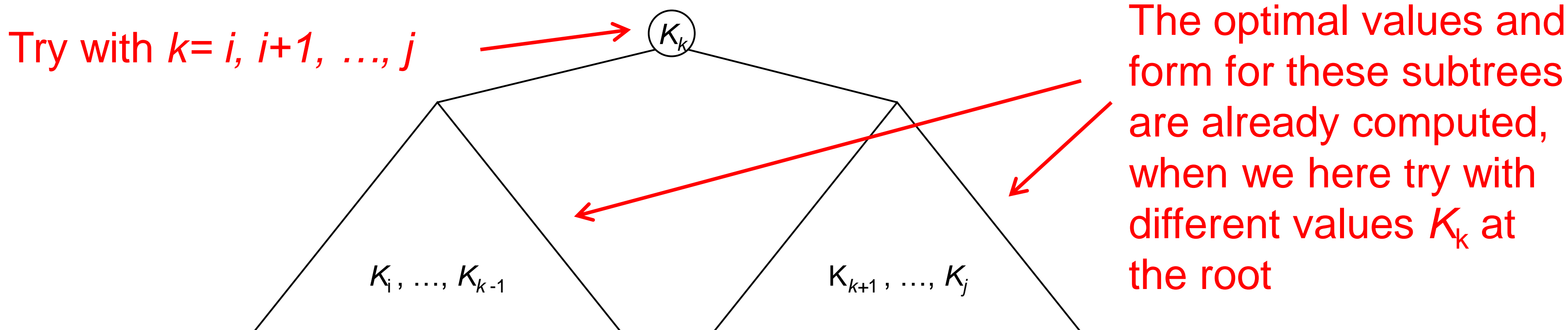
The sum of the p's and q's is 1

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

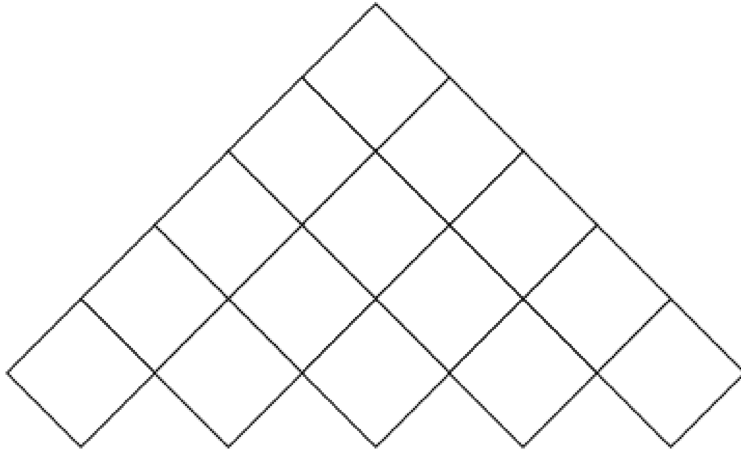
Optimal search trees

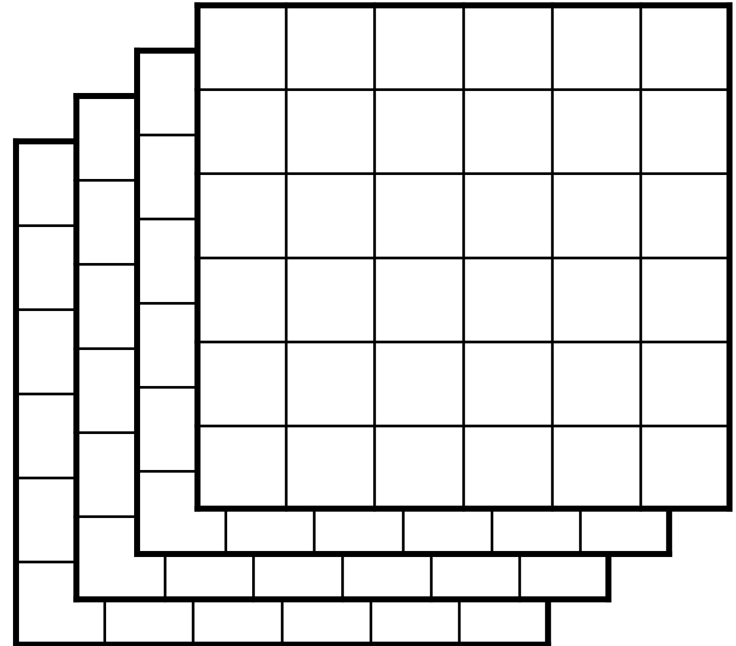
- To get a managable problem that still catches the essence of the general problem, we shall assume that all q-es are zero (that is, we never search for values not in the tree)
- A key to a solution is that a subtree in a search tree will always represent an interval of the values in the tree in sorted 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 $K_i, \ldots, K_j$ we can simply try with each of the values $K_i, \ldots, K_j$ 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.

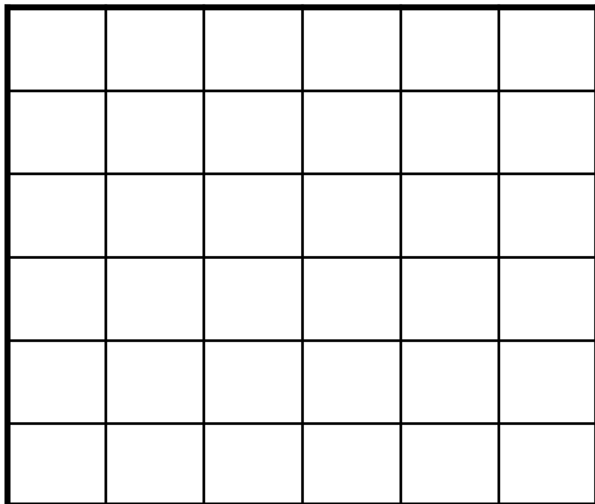


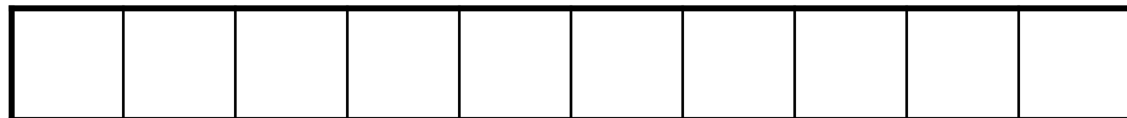
Dynamic programming in general:

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



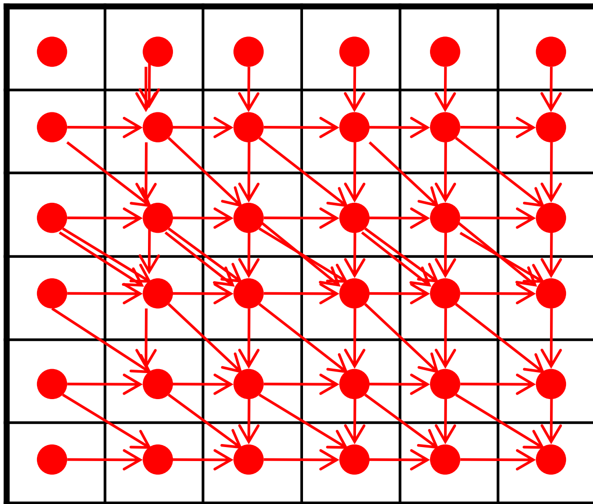
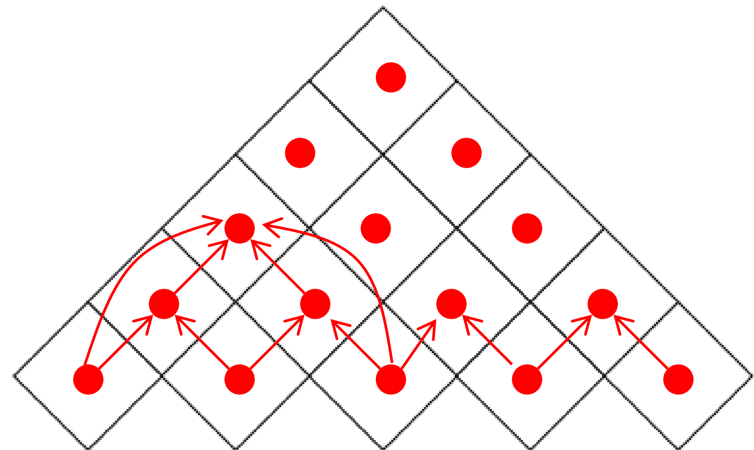






Dynamic programming
Filling in the tables

- 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 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 (see earlier slide).

Dynamic Programming
using memoization

«Top-Down» dynamic programming (Memoization)

- A drawback with bottom up dynamic programming is that you 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