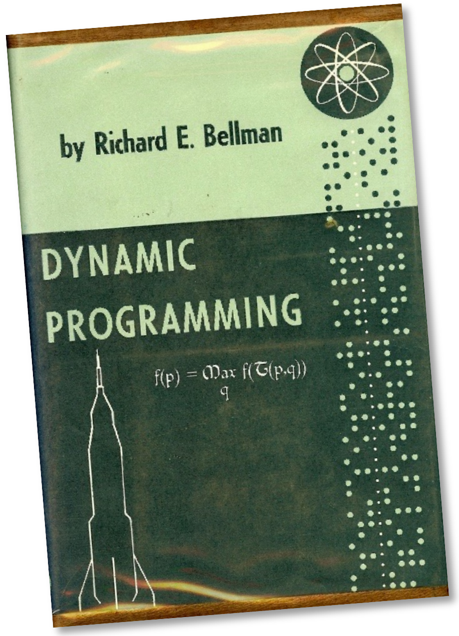
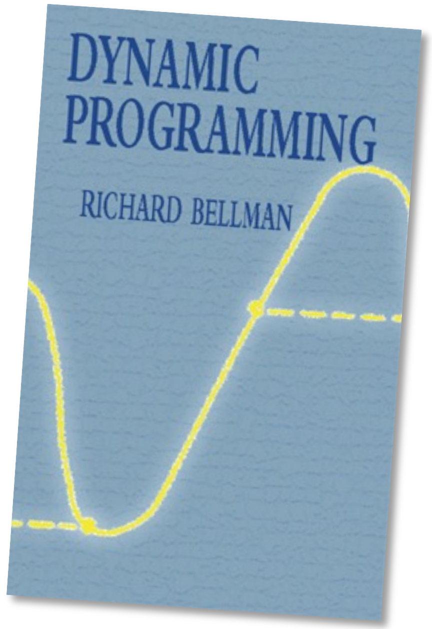
Dynamic Programming

14th september 2022





Dynamic Programming

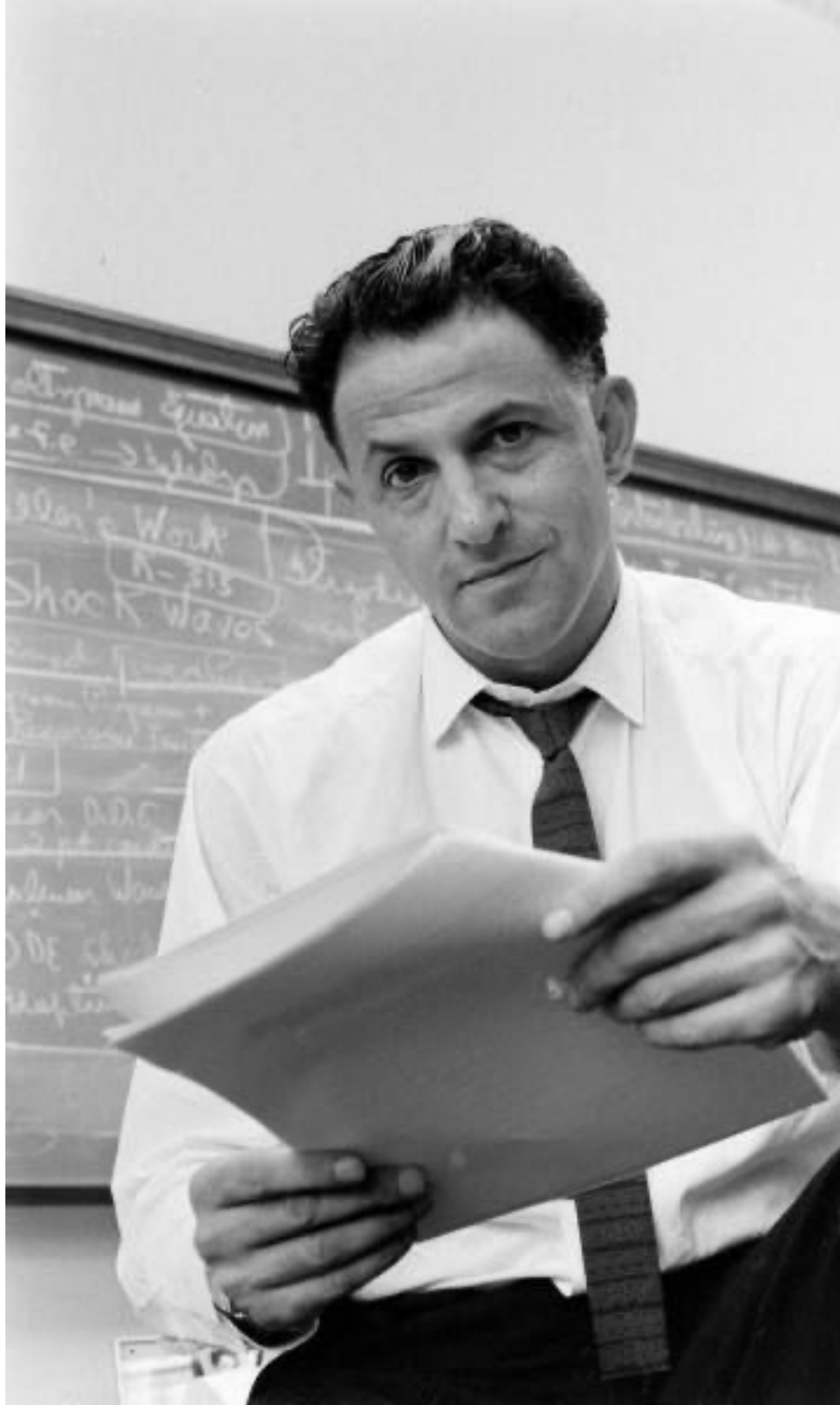
- In Erickson: Ch.3
- (In the B&P: Ch. 9, and Section 20.5)

- These slides have a different introduction to this topic than the books
  - The introduction in B&P is a bit confusing, the formulation of the **principle of optimality** (def. 9.1.1) should be *the other way around*!
  - Erickson is better

- The slides have a lot of text
  - Meant to be a slide set that can be read afterwards

Dynamic Programming

- Dynamic programming was formalised by Richard Bellmann (RAND Corporation) in the 1950s.
- “Programming” should here be understood as planning, scheduling, or making decisions. **It has nothing to do with writing code.**
- “Dynamic” should indicate that it is a stepwise process.

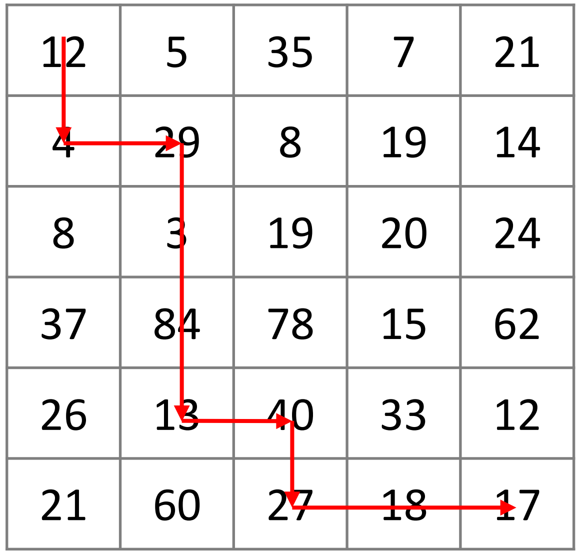


A Simple Example

We are given a matrix W with positive “weights” in each cell:

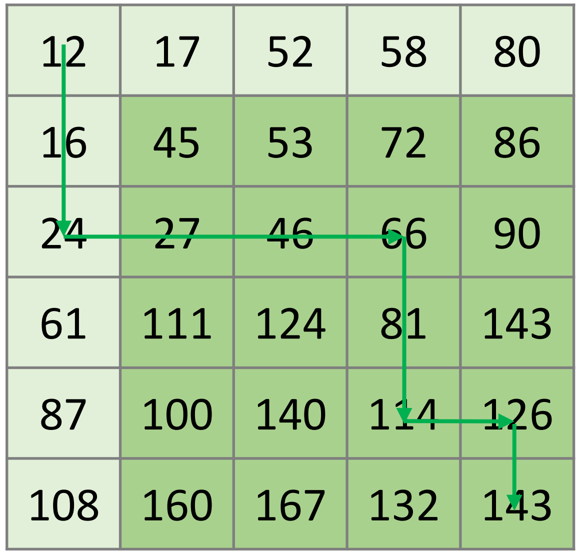
- **Problem:** Find the “best” path (lowest sum of weights) from upper left to lower right corner
- **NB:** The shown **red** path is radomly chosen, and is probably not the best path (has total weight = 255)

- **Solution:** Use a new matrix P to store intermediate results:
  - P[i,j] = The weight of the best path from the start (upper left) to cell [i,j]
  - The formula (recurrence relation) we us will be:
    P[i, j] = min(P[i-1, j], P[i, j-1]) + W[i,j]



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

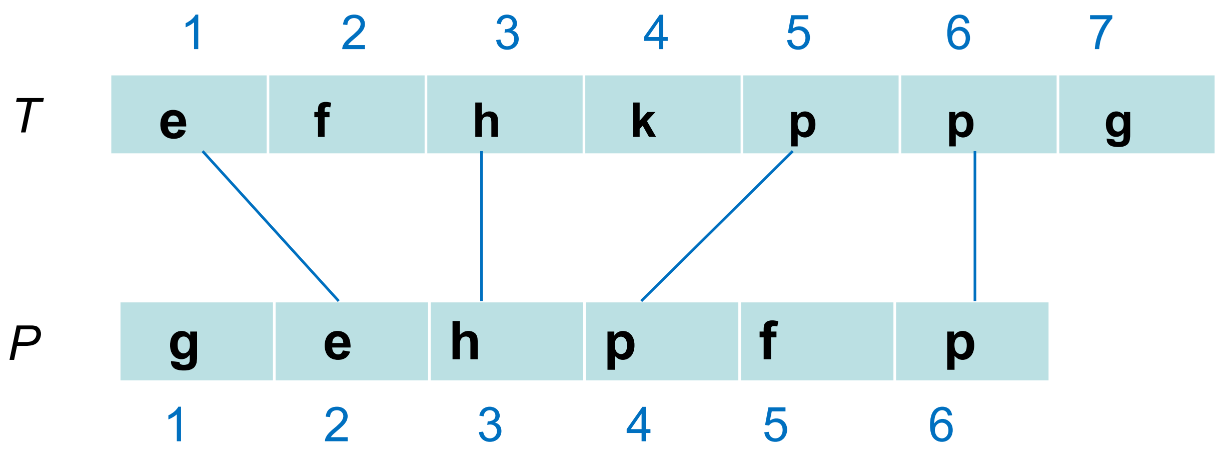
(Initalization as light green)



12	17	52	58	80
16	45	53	72	86
24	27	46	66	90
61	111	124	81	143
87	100	140	114	126
108	160	167	132	143

Longest Common Subsequence (LCS), Ch 9.4

Find the **Longest Common Subsequence** of two strings: P (pattern) and T (text) (not necessarily consecutive)



- Here it is ehpp, the length is 4

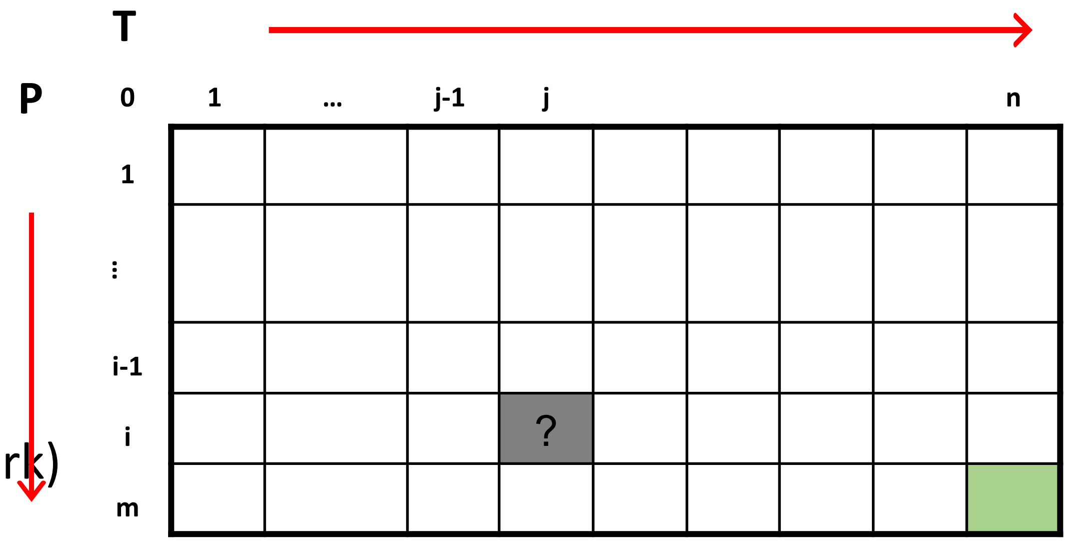
An Idea for Finding LCS

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

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

- Our plan is then to systematically fill in this table so that
  L[i, j] = LCS(P[1:i], T[1:j])
- The value we are looking for, LCS(P, T), will then occur in L[m, n]

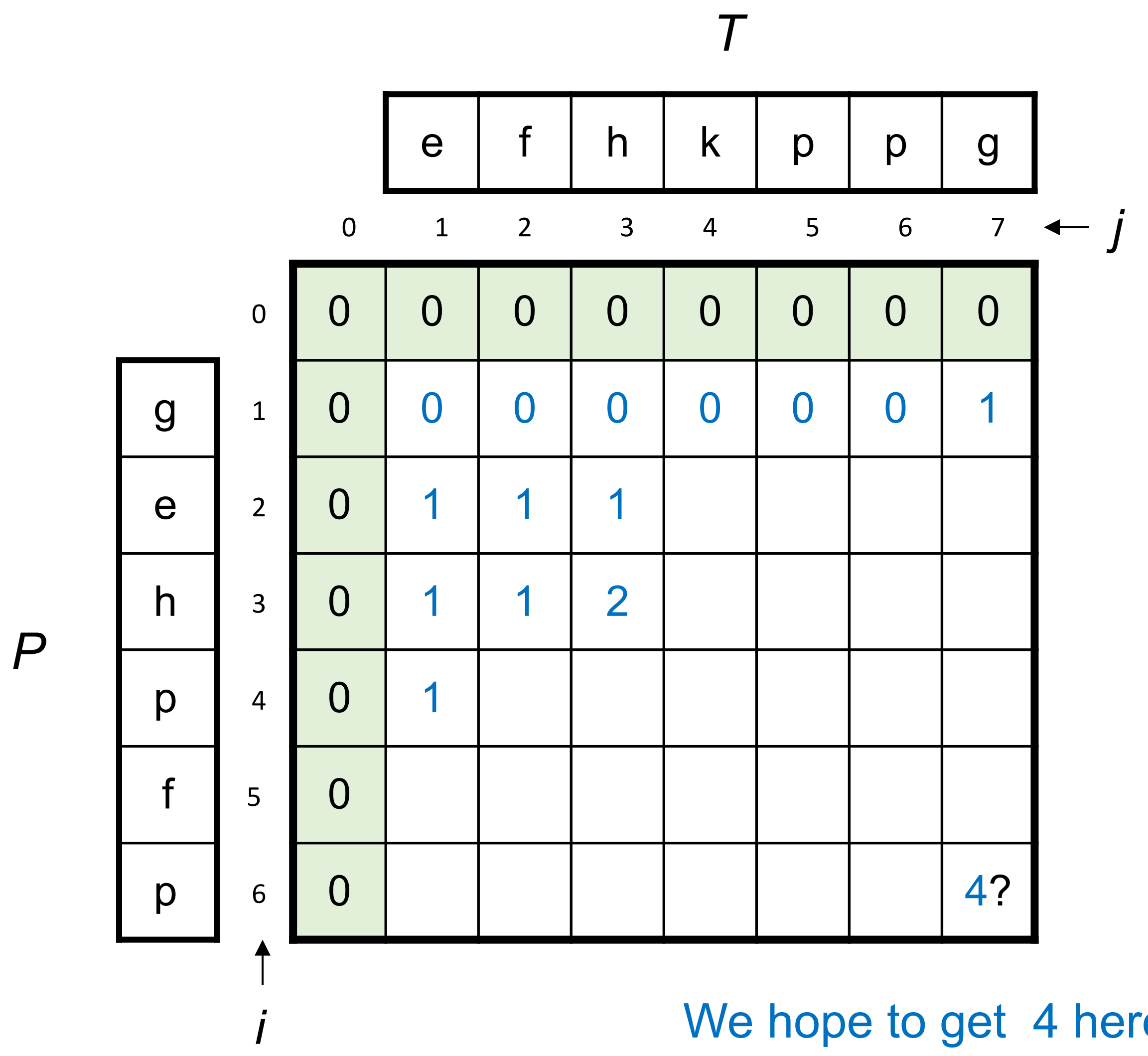
We will do this from smaller to larger index pairs (i, j), by taking row after row from top to bottom (but column after column from left to tight would also work)



Example: P = **gehpfp** and T = **efhkppg**

- We initialize the leftmost column and the topmost row as shown
    - Why is it correct with zeroes here?
    - Note that these celles correspond to the empty prefix of P and/or the empty prefix of T

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



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

We hope to get 4 here!

The General Formula for Filling in L

- We want to calculate L[i,j], and assume we have already computed

L[i-1,j-1]	L[i-1, j]
L[i, j-1]	Find: L[i,j]

Case 1:
If P[i] = T[j] then
L[i, j] = L[i-1,j-1] + 1
WHY?

Case 2:
If P[i] ≠ T[j] then
L[i, j] = max (L[i, j-1], L [i-1, j])
WHY?

T

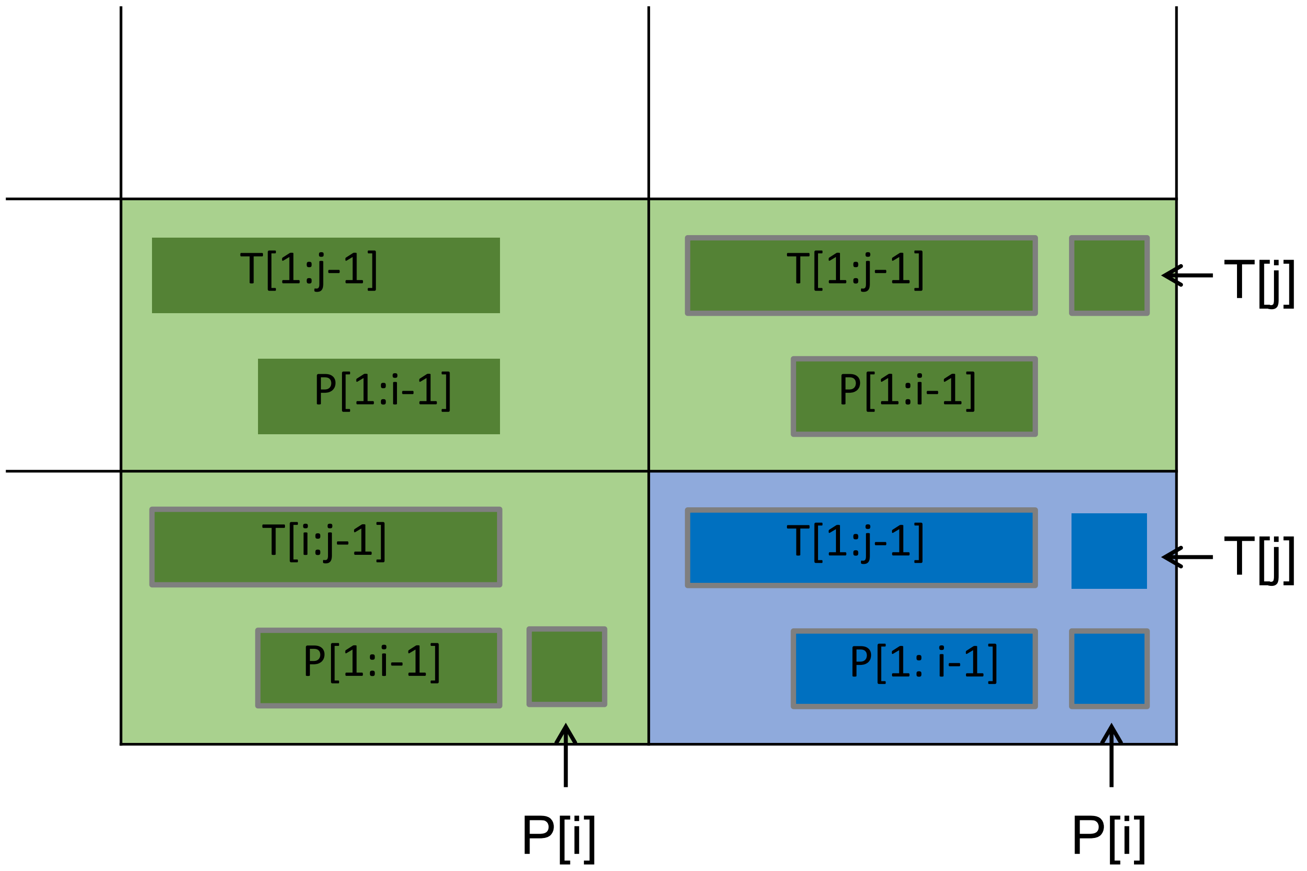
P	↑ *i*		e	f	h	k	p	p	g
		0	1	2	3	4	5	6	7 ← *j*
	0	0	0	0	0	0	0	0	0
g	1	0	0	0	0	0	0	0	1
e	2	0	1	1	1	1	1	1	1
h	3	0	1	1	2				
p	4	0							
f	5	0							
p	6	0							4?

The General Formula for Filling in L

L[i-1,j-1]	L[i-1, j]
L[i, j-1]	Find: L[i,j]

Case 1:
If P[i] = T[j] then
L[i, j] = L[i-1,j-1] + 1

Case 2:
If P[i] ≠ T[j] then
L[i, j] = max (L[i, j-1], L [i-1, j])



Using the General Formula to Fill in L

- We want to calculate L[i,j], and assume we have already computed

	L[i-1,j-1]	L[i-1, j]
	L[i, j-1]	Find: L[i,j]

Case 1:
If P[i] = T[j] then
L[i, j] = L[i-1,j-1] + 1
WHY?

Case 2:
If P[i] ≠ T[j] then
L[i, j] = max (L[i, j-1], L [i-1, j])
WHY?

T

			e	f	h	k	p	p	g	
		0	1	2	3	4	5	6	7	← *j*
	0	0	0	0	0	0	0	0	0	
g	1	0	0	0	0	0	0	0	1	
e	2	0	1	1	1	1	1	1	1	
h	3	0	1	1	2	2	2	2	2	
p	4	0	1	1	2	2	3	3	3	
f	5	0	1	2	2	2	3	3	3	
p	6	0	1	2	2	2	3	4	**4**	

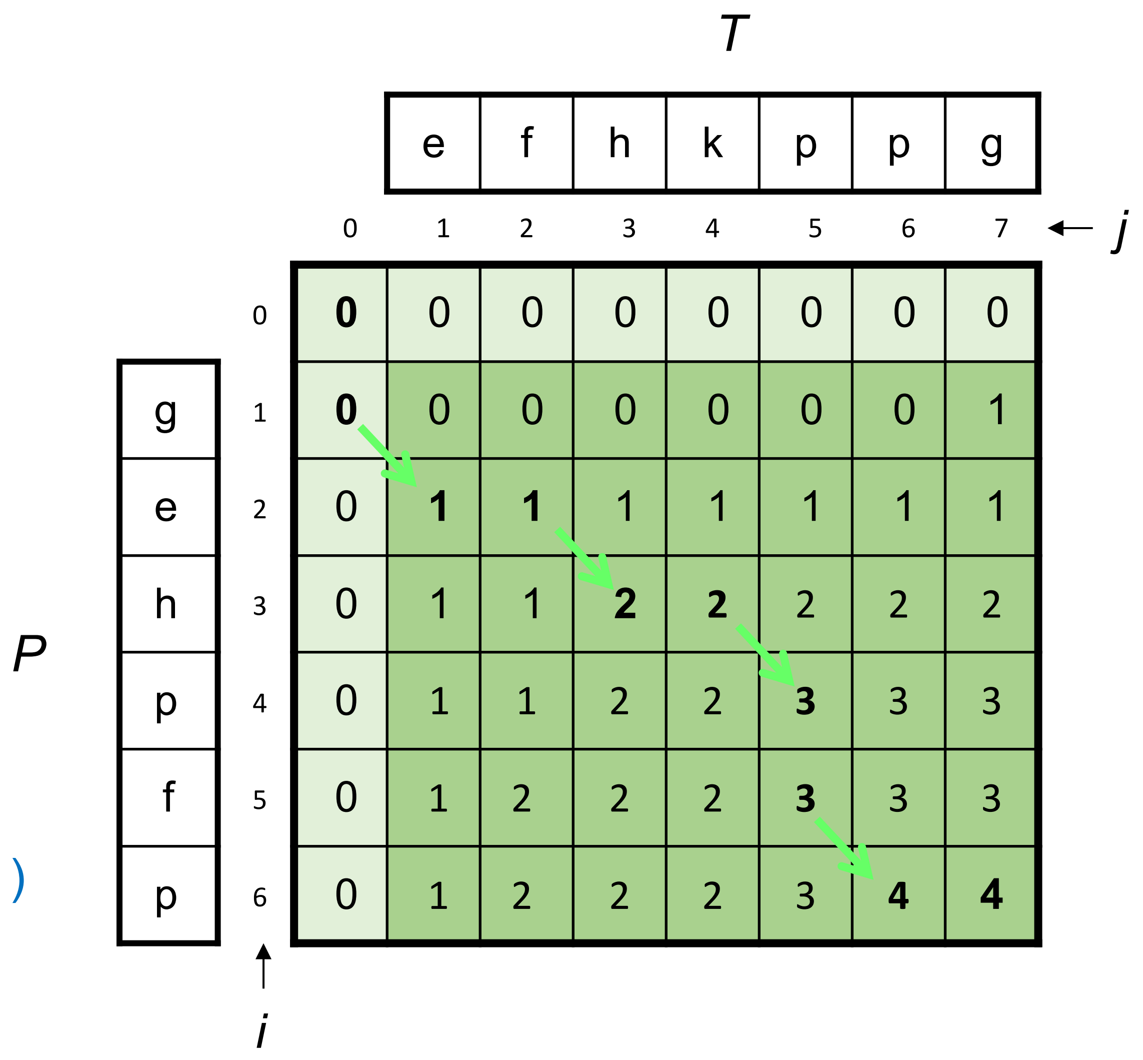
P

↑
i

YAY !!

Finding the Actual Common Subsequence

- To find the actual subsequence (not just the length of it), we trace backwards from L[m,n], highlighting what "caused" each value.
- Arrows indicate the letters included in the Longest Common Subsequence (ehpp)



			e	f	h	k	p	p	g
		0	1	2	3	4	5	6	7
	0	**0**	0	0	0	0	0	0	0
g	1	**0**	0	0	0	0	0	0	1
e	2	0	**1**	**1**	1	1	1	1	1
h	3	0	1	1	**2**	**2**	2	2	2
p	4	0	1	1	2	2	**3**	3	3
f	5	0	1	2	2	2	**3**	3	3
p	6	0	1	2	2	2	3	**4**	**4**

Case 1:
If P[i] = T[j] then
L[i, j] = L[i-1,j-1] + 1
WHY?

Case 2:
If P[i] ≠ T[j] then
L[i, j] = max (L[i, j-1], L [i-1, j])
WHY?

Approximate String Matching (Ch 20.5)

Find strings **similar** to P (pattern) in T (text)

P: u t t x v

T: b s u t t v r t o x i g u t t v x l b t s k u t t z x v k l v 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 similarity?

- We look at how to define and find:
  - The **Edit Distance** between strings

Edit Distance Bewtween Strings

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

Substitution:	One symbol in P is changed to another symbol
Addition:	A new symbol is inserted somwhere in P
Removing:	One symbol is removed from P

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

The smallest number of such operations needed to convert *P* to *T*
(or T to P, it is symmetric)

$\underset{P}{\text{logarithm}} \rightarrow \text{alogarithm} \rightarrow \text{algarithm} \rightarrow \underset{T}{\text{algorithm}}$ (Steps: +a, -o, a->o)

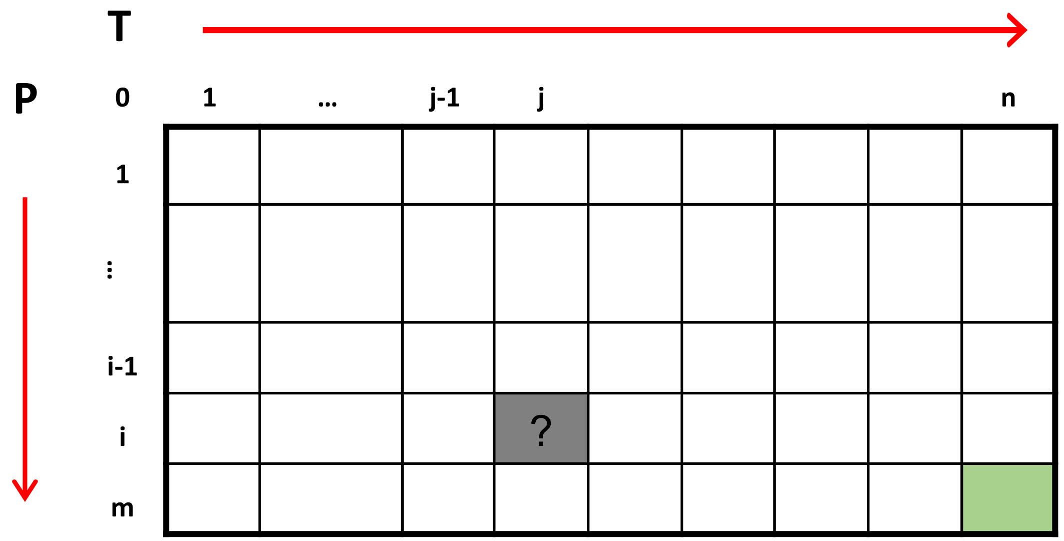
ED(”logarithm”, ”algorithm”) = 3   There are no shorter ways!of doing it

An Idea for Calculating Edit Distance

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

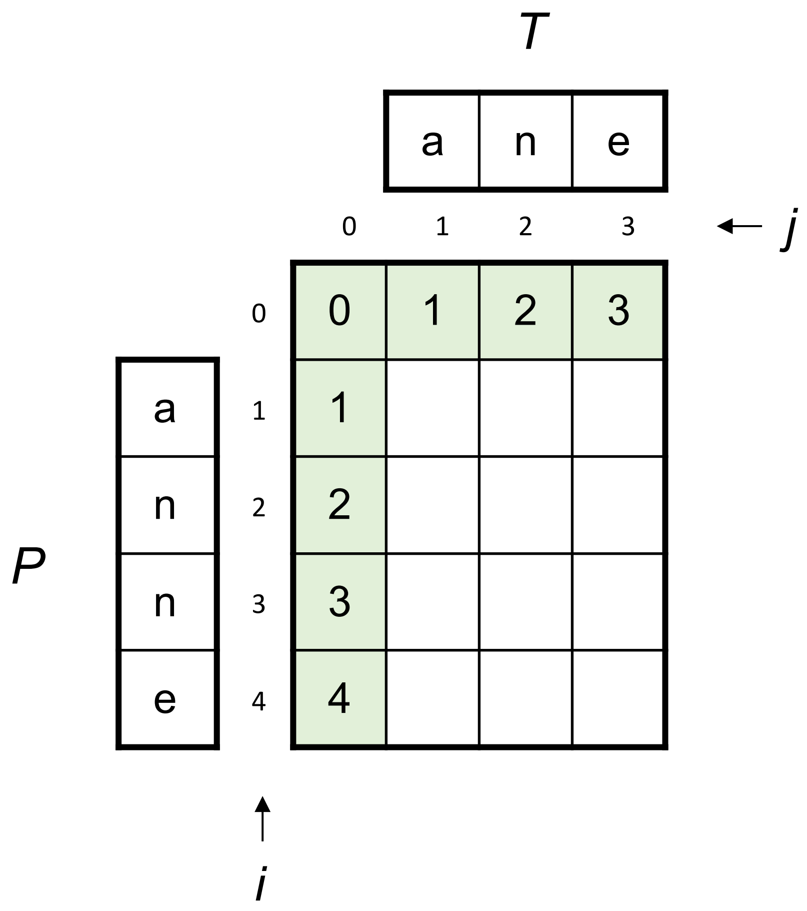
- Our plan is again to systematically fill in this table, but now so that
  D[i, j] = **Edit Distance** between the strings P[1:i] and T[1:j]
- The value we are looking for, ED(P, T), will then occur in D[m, n]

Again, we will do this from smaller to larger index pairs (i, j), by taking row after row (but column after column would also work)



Example: P = **anne** and T = **ane**

- We initialize the leftmost column and the topmost row as shown
  - Why is this initaialization correct?
  - Note that these celles correspond to the empty prefix of P and/or the empty prefix of T
    D[i, j] = ED(P[1:i], T[1:j])



$i \backslash j$	0	1	2	3
0	0	1	2	3
1	1			
2	2			
3	3			
4	4			

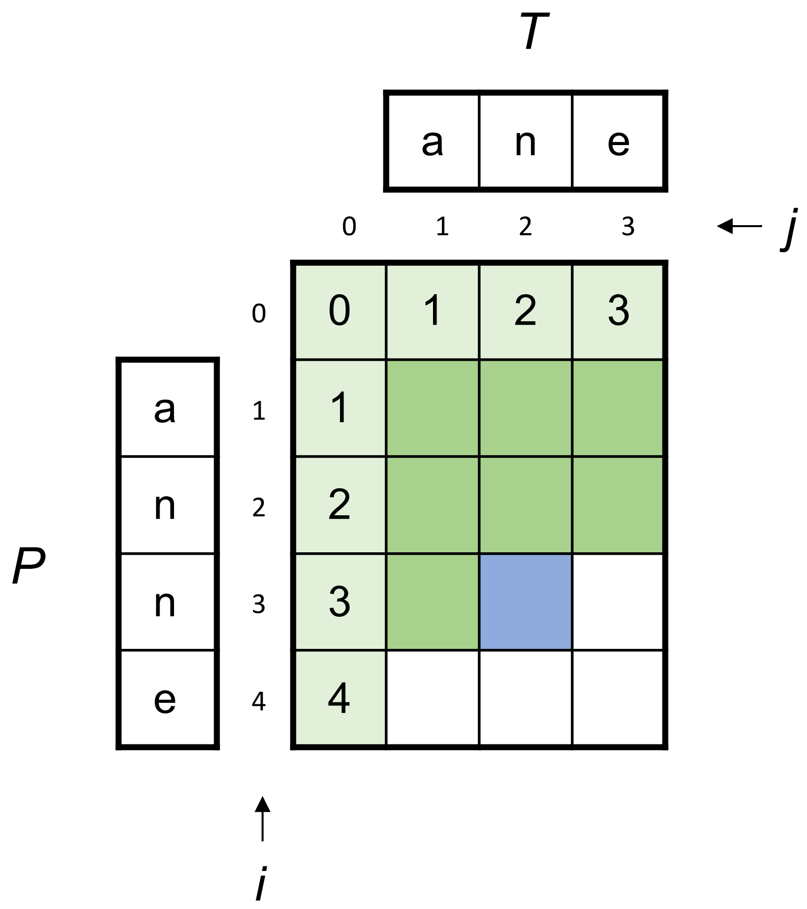
Example: P = **anne** and T = **ane**

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 above and to the left

We first assume that P[i] = T[j]

- We know that P[1: i-1] can be transformed to T[1:j-1] in D[i-1, j-1] steps
- Therfore **P[1:i-1]** ⊢ **n** can also be transformed into **T[1:j-1]** ⊢ **n** in D[i-1, j-1] steps

- **So If P[i] = T[j], then D[i, j] = D[i-1,j-1]**



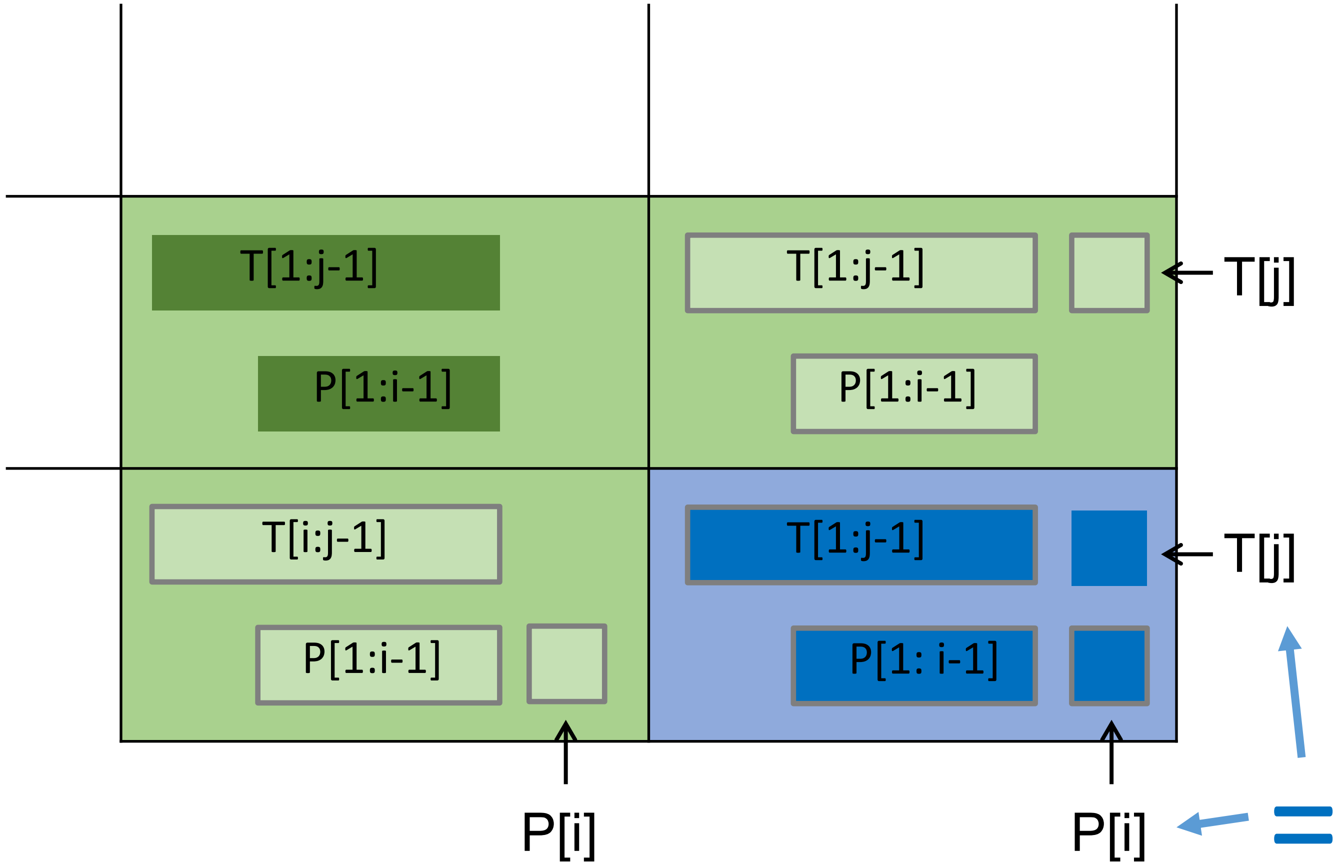
Example: P = **anne** and T = **ane**

D[i-1,j-1]	D[i-1, j]
D[i, j-1]	Find: D[i,j]

Case 1:

If P[i] = T[j] then

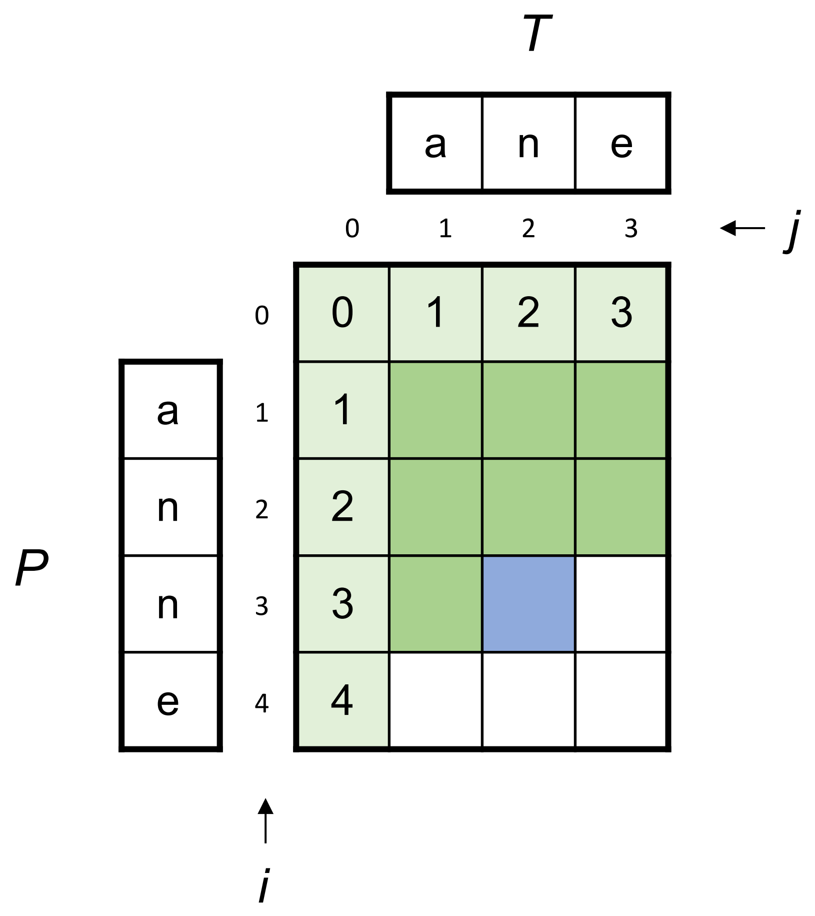
D[i, j] = D[i-1,j-1]



Example: P = **anne** and T = **ane**

We next assume that P[i] ≠ T[j]

- We know that P[1: i-1] can be transformed to T[1:j-1] in D[i-1, j-1] steps
- Therfore **P[1: i-1]** ⊦ **x** can be transformed into **T[1: j-1]** ⊦ **y** in D[i-1, j-1] + 1 steps (substitution)
- Likewise **P[1: i-1]** ⊦ **y** can be transformed into **T[1: j-1]** in D[i, j-1] +1 steps (remove y from P)
- And **P[1: i-1]** can be transformed into **T[1:j-1]** ⊦ **x** in D[i-1, j] + 1 steps (insert x in P (same as remove x from T)
- **So if P[i] ≠ T[j] then**
  D[i, j] = min(D[i-1,j-1], D[i,j-1], D[i-1,j]) + 1



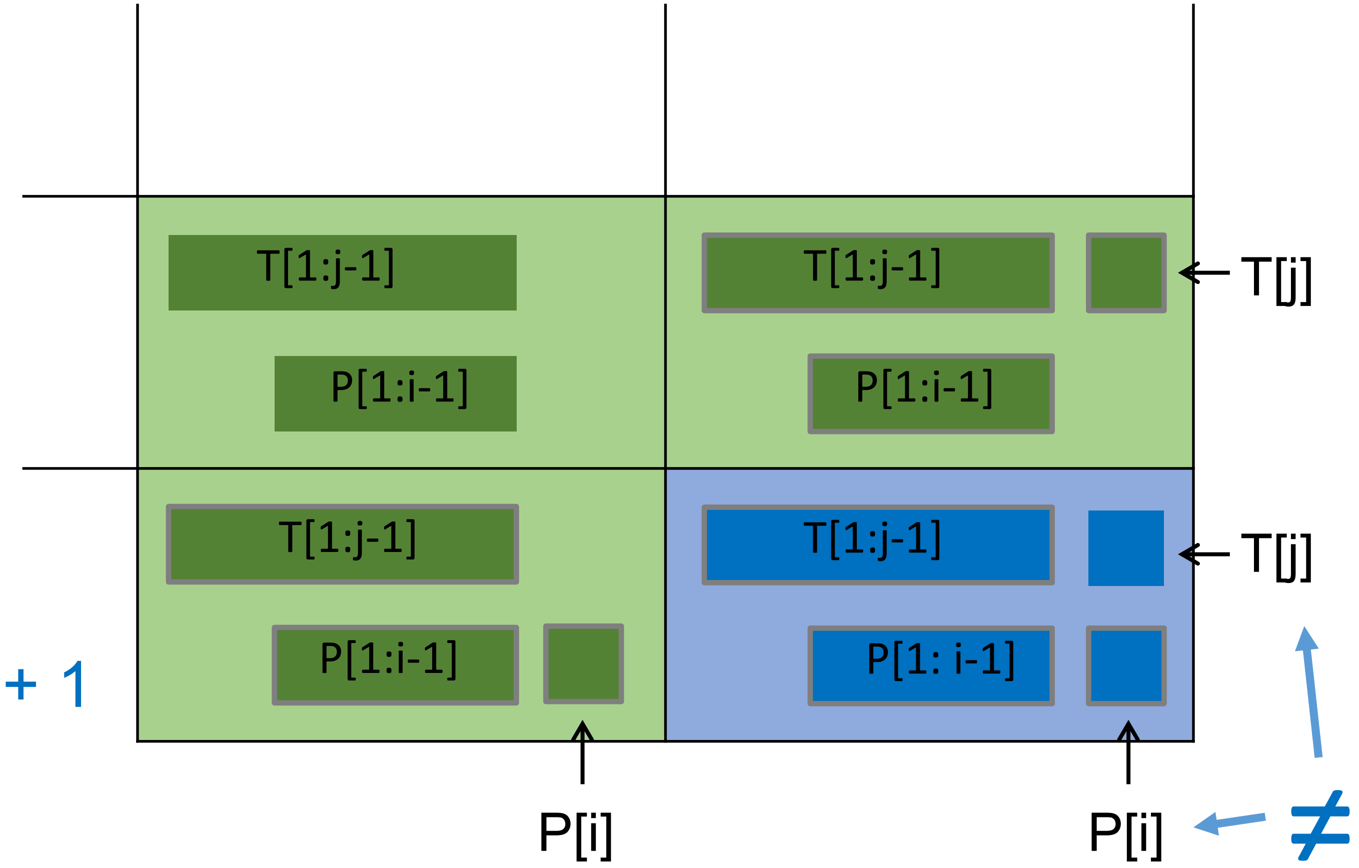
Example: P = **anne** and T = **ane**

	D[i-1,j-1]	D[i-1, j]
	D[i, j-1]	Find: D[i,j]

Case 2:

If P[i] ≠ T[j] then

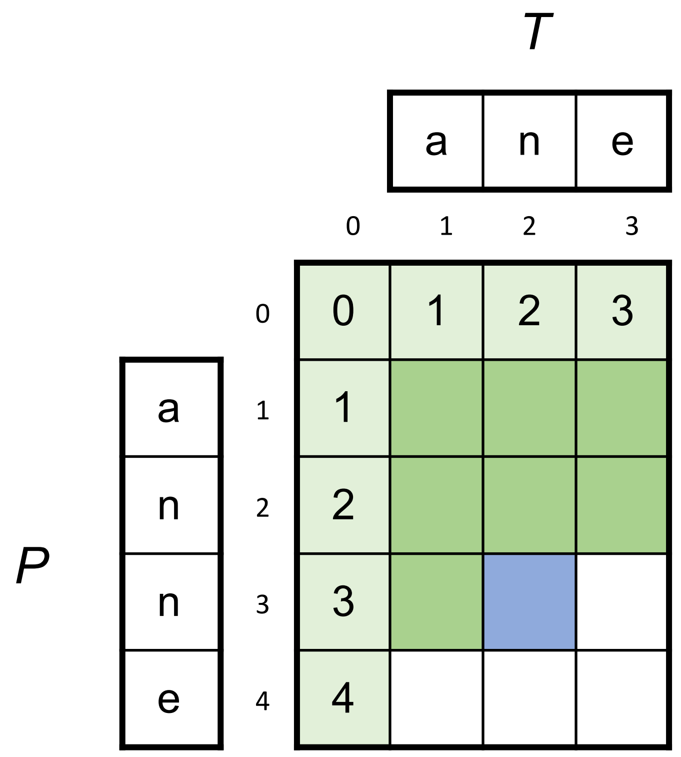
D[i, j] = min(D[i-1,j-1], D[i,j-1], D[i-1,j]) + 1



General formula for filling in D

$$D[i, j] = \begin{cases} D[i, j] = D[i-1,j-1] & \text{if } P[i] = T[j] \\ D[i, j] = \min(\underbrace{D[i-1,j-1]}_{\text{substitution make } P[i] = T[j]}, \underbrace{D[i,j-1]}_{\text{addition in P Same as removal from T}}, \underbrace{D[i-1,j]}_{\text{removal from P}}) + 1 & \text{else} \end{cases}$$

$D[0, 0] = 0, D[i,0] = D[0,i] = i$ Initialization



		T	a	n	e
P		0	1	2	3
	0	0	1	2	3
a	1	1	🟩	🟩	🟩
n	2	2	🟩	🟩	🟩
n	3	3	🟩	🟦	
e	4	4			

General formula for filling in D

$$D[i, j] = \begin{cases} D[i, j] = D[i-1,j-1] & \text{if } P[i] = T[j] \\ D[i, j] = \min(D[i-1,j-1], D[i,j-1], D[i-1,j]) + 1 & \text{else} \end{cases}$$

substitution make P[i] = T[j]

addition in P Same as removal from T

removal from P

$D[0, 0] = 0, D[i,0] = D[0,i] = i$ Initialization

T

P			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**

A Program for Calculating Edit Distance

```
function EditDistance ( P [1:n ], T [1:m ] )
    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, after the initialization, we look at the pairs (i, j) in the following order (line after line):

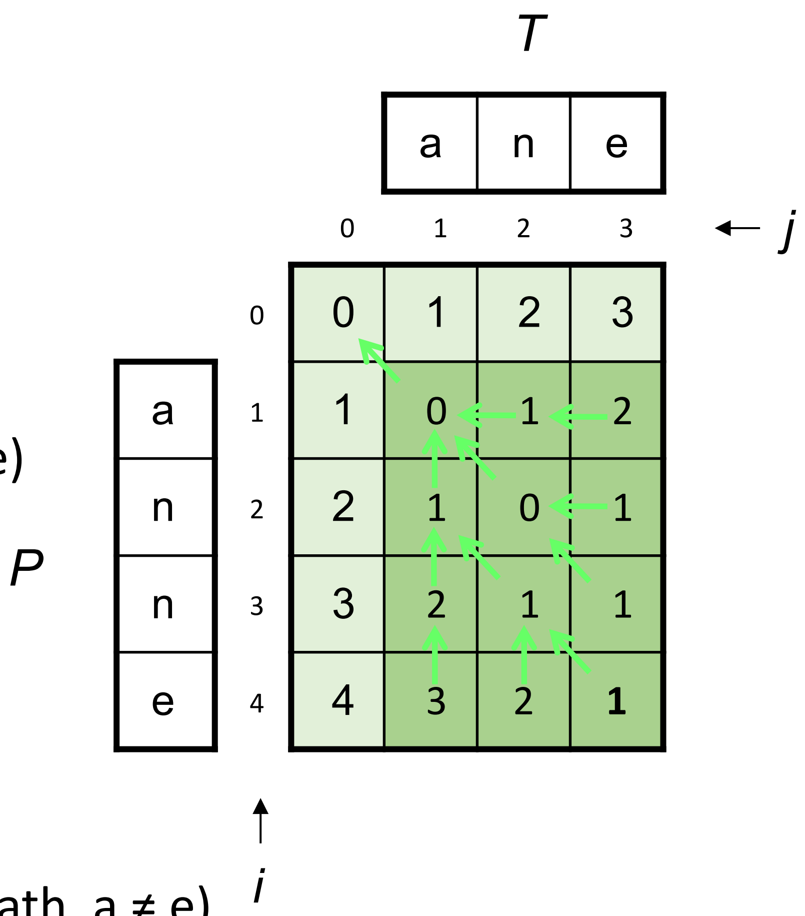
(1,1) (1,2) … (1,n)
(2,1) (2,2) … (2,n)
…
(m,1) …   (m,n)

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

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

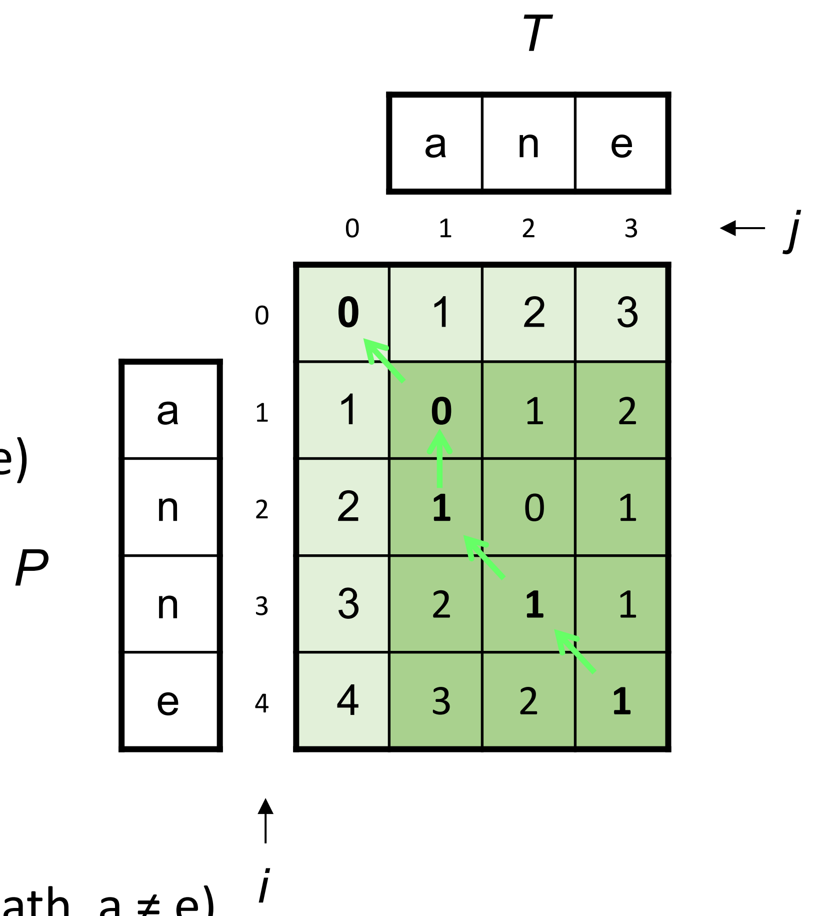
Finding the Edit Steps

- Arrows indicate how we calculated each value
  - ↖ Diagonally and P[i] = T[j]
    - No edit needed (e.g. D[3,2], n = n)
  - ↖ Diagonally and P[i] ≠ T[j]
    - Substitution (e.g. D[3,3], not used in final edit path, n ≠ e)
  - ↑ Upwards (and thus P[i] ≠ T[j])
    - A letter is deleted from P (e.g. D[2,1], n ≠ a)
  - ← To the left (and thus P[i] ≠ T[j])
    - A letter is added to P (e.g. D[1,3], not used in final edit path, a ≠ e)



Finding the Edit Steps

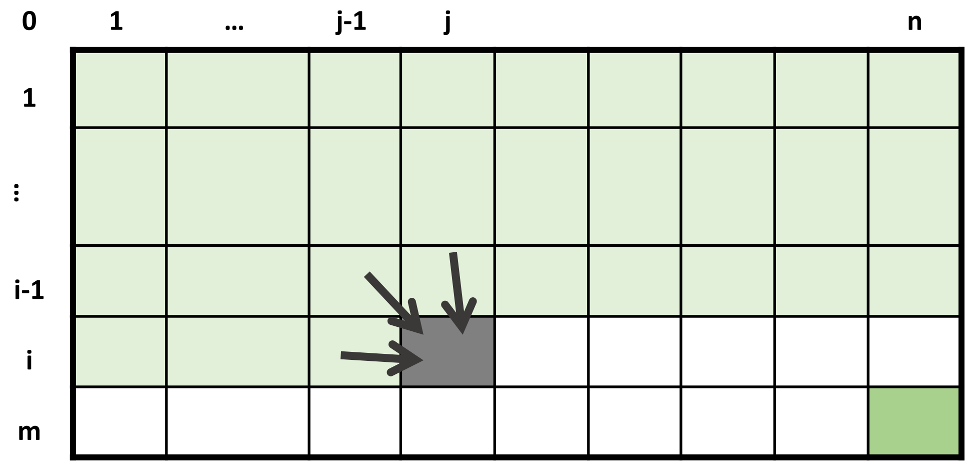
- Arrows indicate how we calculated each value
  - ↖ Diagonally and P[i] = T[j]
    - No edit needed (e.g. D[3,2], n = n)
  - ↖ Diagonally and P[i] ≠ T[j]
    - Substitution (e.g. D[3,3], not used in final edit path, n ≠ e)
  - ↑ Upwards (and thus P[i] ≠ T[j])
    - A letter is deleted from P (e.g. D[2,1], n ≠ a)
  - ← To the left (and thus P[i] ≠ T[j])
    - A letter is added to P (e.g. D[1,3], not used in final edit path, a ≠ e)



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

About Dynamic Programming in General

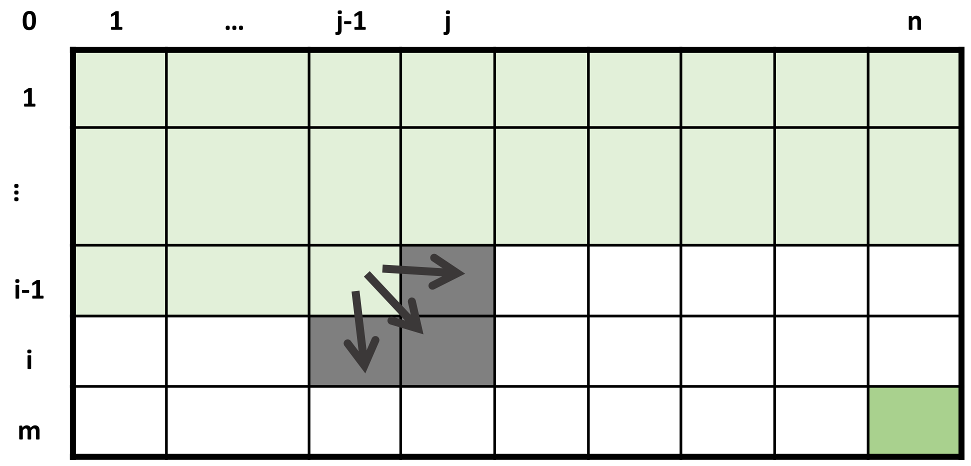
- Dynamic Programming is typically used to solve optimazation problems.
- The instances of the problem must be handeled from smaller to larger ones, and the smallest (or simplest) instances can usually easily be solved directly (and be used for initialization of a program)
- For each problem instance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



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

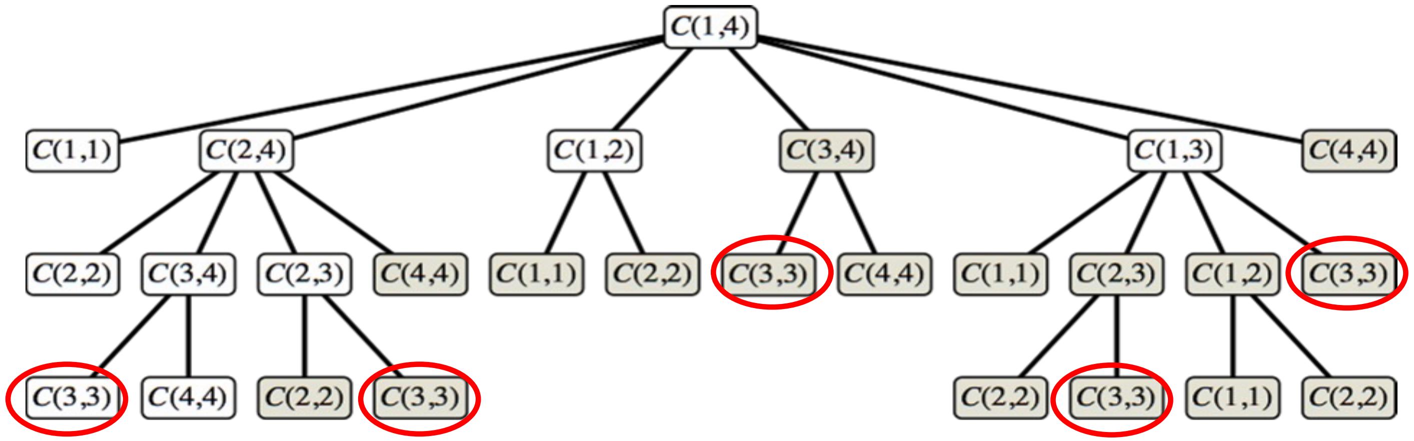
When Should We Use Dynamic Programming?

- 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 suitabel 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



Another (Slightly Abstract) Exmple

- As indicated on the previous slide, Dynamic Programming is more useful if the solution to a certain instance is used in the solution of many (larger) instances (assuming that the size of an instanstance C(i, j) is j – i)
- In the problem C below, an instance is given by some data (e.g two strings) and by two intergers *i* and *j.* Assume the corresponding instances are 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 here j – i).
- Below, the children of a node *N* indicate the instances that we need the solution of, to compute the solution to instance *N*. We would therefore get this tree if we use recursion without remembering computed values at all.



A Rather Formal Basis for Dynamic Programming (not central to IN 3130)

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
- 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 (Trad.) and Top-Down (Memoization)

1. **Traditional Dynamic Programming (bottom up)**

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

2. **"Top-Down" Dynamic Programming - Memoization**

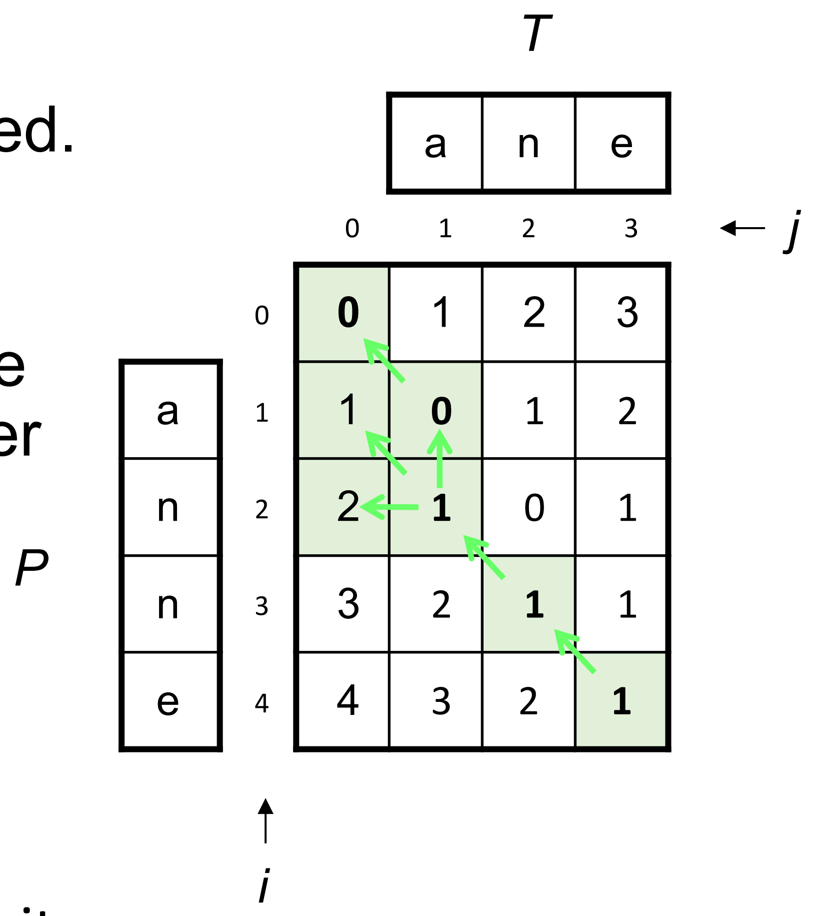
- A drawback with traditional dynamic programming is that one usually solves a number of smaller instances that turn out not to be neededed for the actual (larger) instance that we are really interested in
- We can instead start at the (large) instance we want to solve, and do the computation recursively top-down. Also here we put computed solutions in the table as soon as they are computed
- Each time we need to find the answer to an instance we first check in the table whether it is already solved, and if so we only use the stored solution. Otherwise we do recursive calls, and store the solution
- The table entries then need a special marker "not computed", which also should be the initial value of the entries

Top-Down Dynamic Programming (Memoization)

1. Start at the instance you want to solve, and ask recursively for the solution to the instances needed. The recursion will follow the green arrows in the figure
2. As soon as you have an answer, store it in a table table, and retrieve it from there when/if the answer to the same instance is needed later

Benefit:
You only have to compute needed table entries

But:
Managing the recursive calls takes some extra time, so it it not always faster.



			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**

Problems typically solved with Dynamic Programming

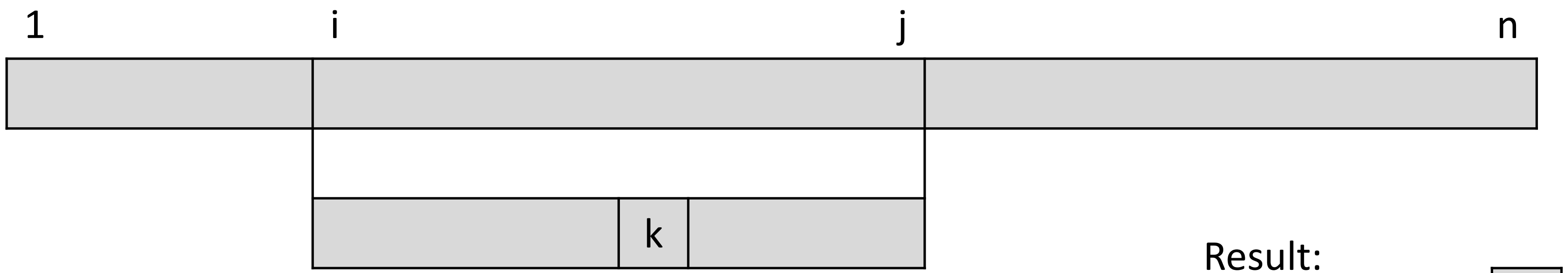
Both the **Optimal Matrix Multiplication** and **Optimal Search Tree** problem are of this type

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

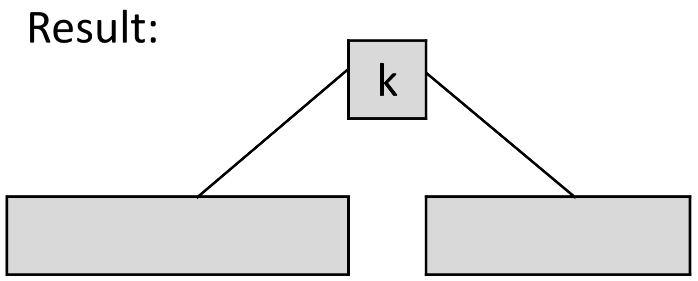
Problems typically solved with Dynamic Programming

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

  Q(i, j) = max over k = i, i+1, …, j of Q'(i, k, j)



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



Optimal Matrix Multiplication Order, Ch 9.2

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

We do this by parenthesizing and multiplying pairs of matrices

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

The multiplication can be done in different ways (different parenthesis structures):

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

The cost (number of scalar multiplications) can vary a lot

Optimal Matrix Multiplication Order, Ch 9.2

Gicen two matrices

A = $p \times q$ matrix
B = $q \times r$ matrix.

The cost (number of scalar multiplications) of computing $A \cdot B$ is $p \cdot q \cdot r$, *and the result is a* $p \times r$ matrix

Example

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

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

Calculating $D = (A \cdot B)$ costs 5,000 and results in a 10×5 matrix
Calculating $D \cdot C$ costs 2,500
Total cost for $(A \cdot B) \cdot C$ is 7,500

Calculating $E = (B \cdot C)$ costs 25,000 and results in a 100×50 matrix
Calculating $A \cdot E$ costs 50,000
Total cost for A · (B · C) is 75,000

Optimal Matrix Multiplication Order, Ch 9.2

- Given a sequence of matrices $M_0, M_1, \ldots, M_{n-1}$ we want to compute the product $M_0 \cdot M_1 \cdot \ldots \cdot M_{n-1}$ as cheaply as possible
  - we must find an optimal parenthesis structure
- A parentherization of the sequence is a partition of the sequence into two sub-sequences that both have to be parenthesized

$$(M_0 \cdot M_1 \cdot \ldots \cdot M_k) \cdot (M_{k+1} \cdot M_{k+2} \cdot \ldots \cdot M_{n-1})$$

  - We must try all *k* to find the place to partition the sequence
  - Each *k gives rise to two subproblems:* parentherization *of the left and right sub-sequence*
- A sequence of one matrix is a parentherization in itself (also obvious with two)

Optimal Matrix Multiplication Order, Ch 9.2

- Let $d_0, d_1, \ldots, d_n$ be the dimensions for the sequence of matrices $M_0, M_1, \ldots, M_{n-1}$ so that M_i has dimensions $d_i \times d_{i+1}$
- Let $m_{i,j}$ be the cost of an optimal parenterization of $M_\mathrm{i}, M_{\mathrm{i}+1}, \ldots, M_j$
- The formula for $m_{i,j}$ will then be as follows :

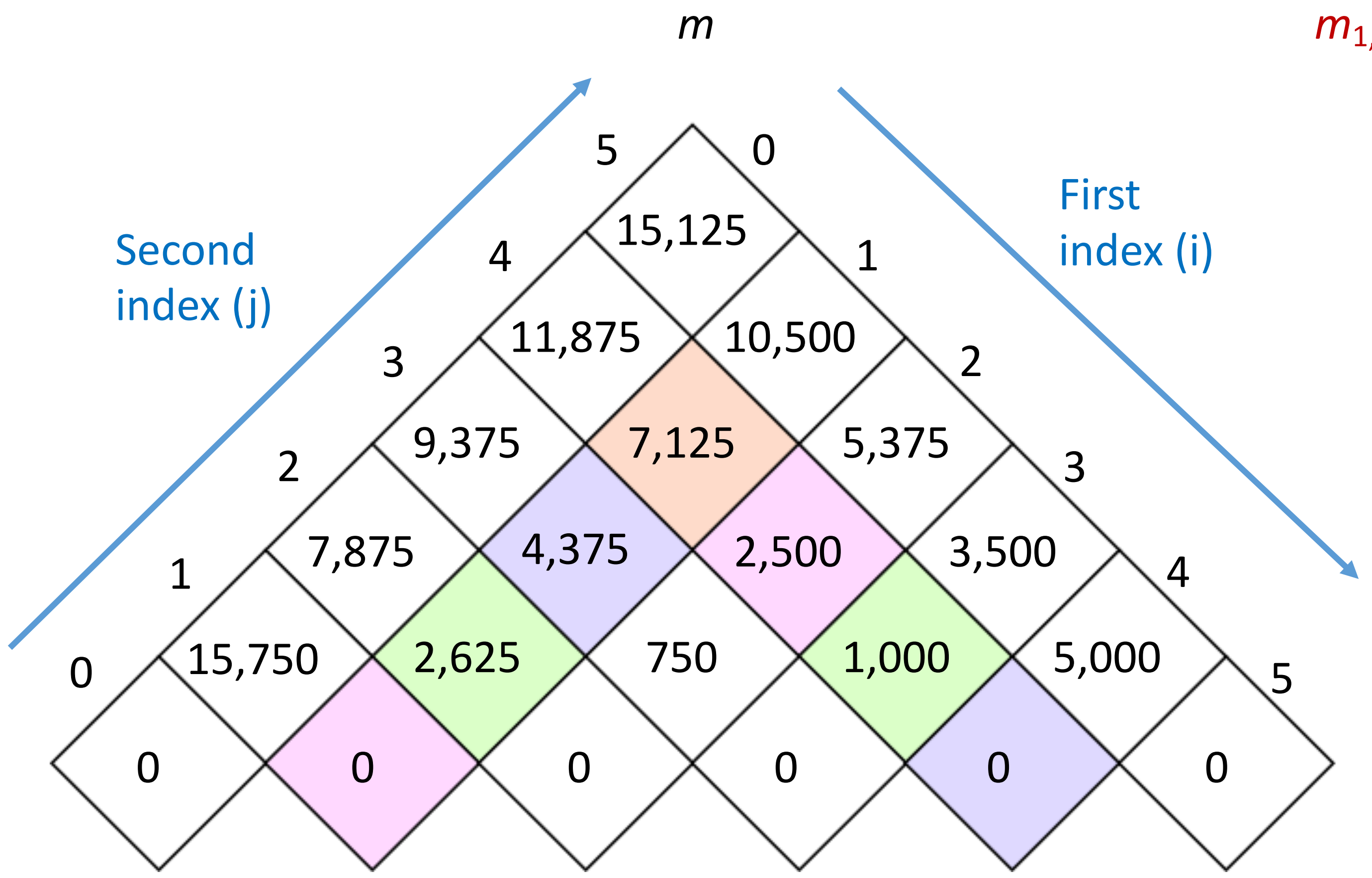
$$m_{i,j} = \min_{i \le k < j} \{ m_{i,k} + m_{k,j} + d_i d_{k+1} d_{j+1} \} \text{ for all } 0 \le i < j < n - 1$$

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

- The total cost is found in $m_{0,n\text{-}1}$

Optimal Matrix Multiplication Order, Ch 9.2

d	30	35	15	5	10	20	25



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

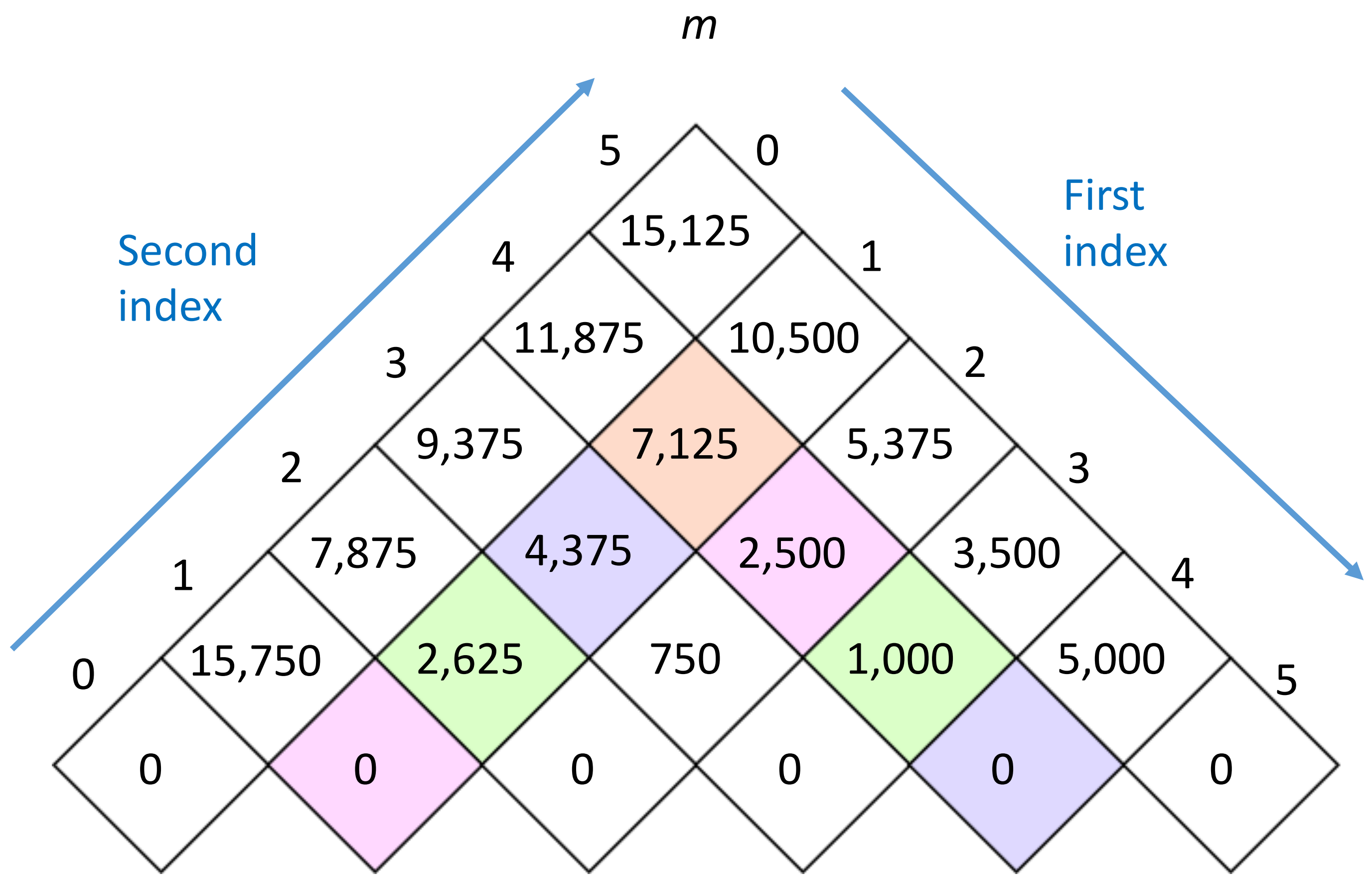
$$\begin{aligned} = \min(&35 \cdot 15 \cdot 20 + 0 + 2{,}500,\\ &35 \cdot 5 \cdot 20 + 2{,}625 + 1{,}000,\\ &35 \cdot 10 \cdot 20 + 4{,}375 + 0) \end{aligned}$$

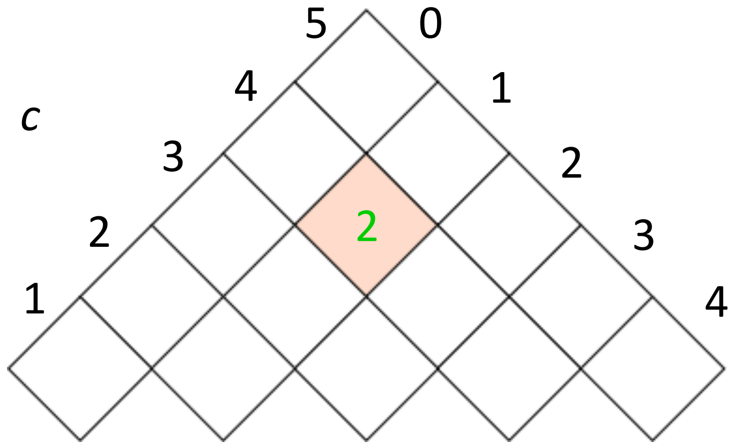
$$= \min(13000, 7125, 11375)$$

$$= 7125$$

Optimal Matrix Multiplication Order, Ch 9.2

d						
	30	35	15	5	10	20	25





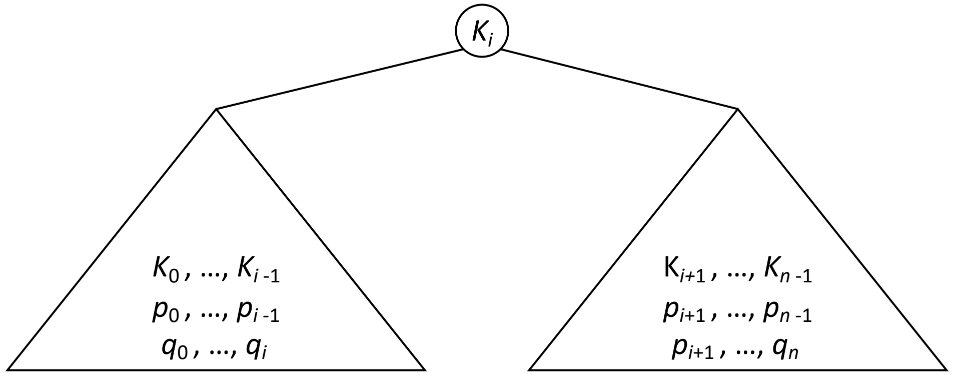
Store the k in a separate table to remember the solution.

Optimal Matrix Multiplication Order, Ch 9.2

```
function OptimalParens( d[0 : n – 1] )
    for i ← 0 to n-1 do
        m[i, i] ← 0
    for diag ← 1 to n – 1 do            // helper variable, we fill in table diagonally
        for i ← 0 to n – 1 – diag do
            j ← i + diag
            m[i, j] ← ∞
            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
```

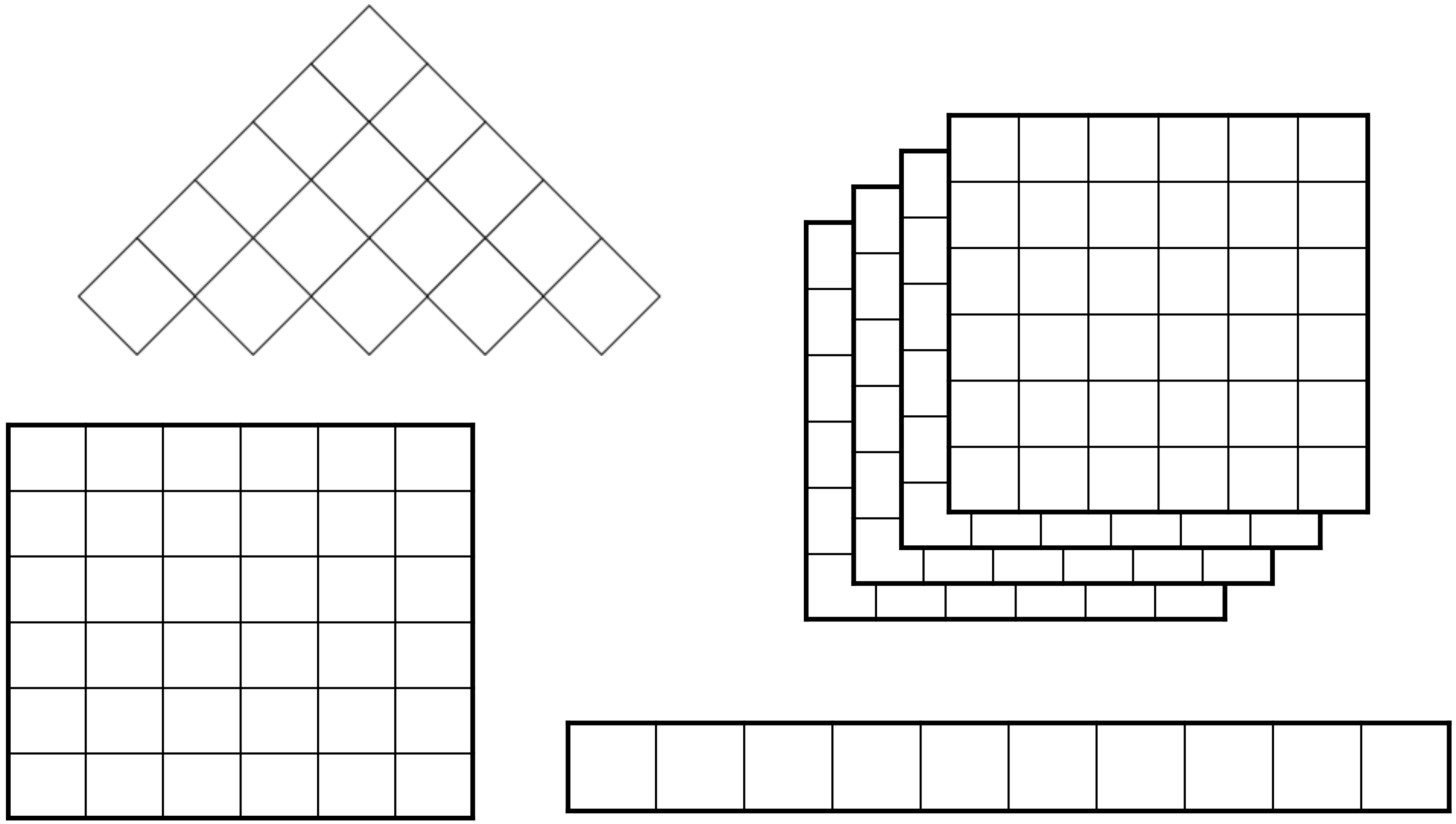
Optimal Search Trees, Ch 9.3

- A binary tree for keys K_0 , … $K_{n\text{-}1}$ conists of a root with key K_i , and two subtrees L and R


K_i
K_0 , …, $K_{i\text{ -}1}$
p_0 , …, $p_{i\text{ -}1}$
q_0 , …, q_i
K_{i+1} , …, $K_{n\text{ -}1}$
p_{i+1} , …, $p_{n\text{ -}1}$
p_{i+1} , …, q_n

- We must try all possible roots K_i to find the best way to partition the tree
    - Each root gives rise to two sub-problems – optimizing the left and rigt subtree

Dynamic Programming in General: Filling in tables bottom-up (smallest instances first)



Dynamic Programming – Calculations

- Normally we solve all small instances before moving on to larger ones
- If we know which small solutions are needed to solve a larger one, we can deviate from the above
- Thus, if we know the **dependency graph** of the problem, we must look at the intances in an order that conforms with that dependency