Approximate pattern matching

Lecture 05.01

by Marina Barsky

Sequence similarity

- The biological sequences encode and reflect higher-level molecular structures and mechanisms

- **In bimolecular sequences (DNA, RNA or protein), high sequence similarity usually implies significant structural and functional similarity**

- A tractable, though partly heuristic way to infer the structure and function of an unknown protein is to search for the similar known proteins at the sequence level

Similar but not identical!

- We are looking for sequences that are similar to each other
- However they are never exactly the same due to small changes accumulated over generations

- How do we define and measure similarity?

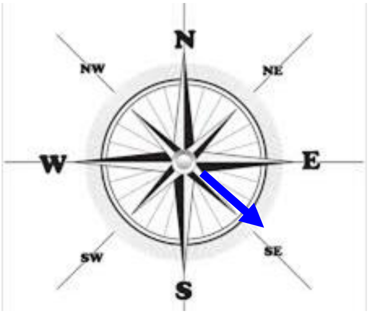
Approximate pattern matching

- Approximate – means some errors are allowed in valid matches
- The shift is accompanied by a shift in technique:

  dynamic programming

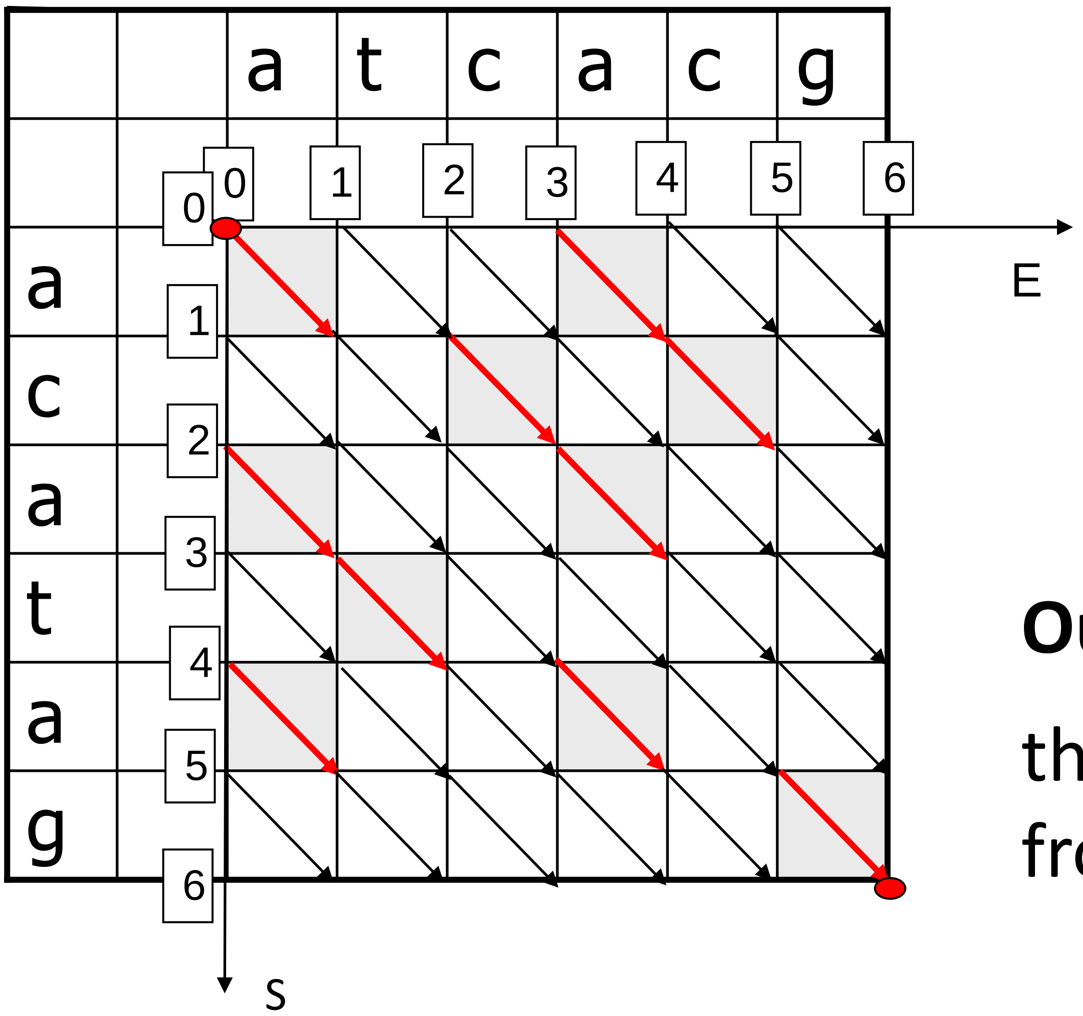
Dynamic Programming

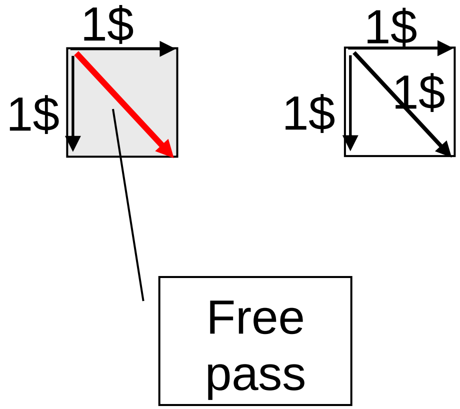
The main tool in approximate pattern matching



Problem: the cheapest path in a special grid

Input:

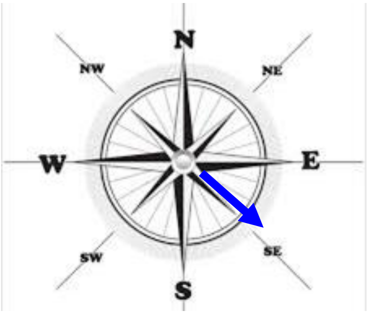


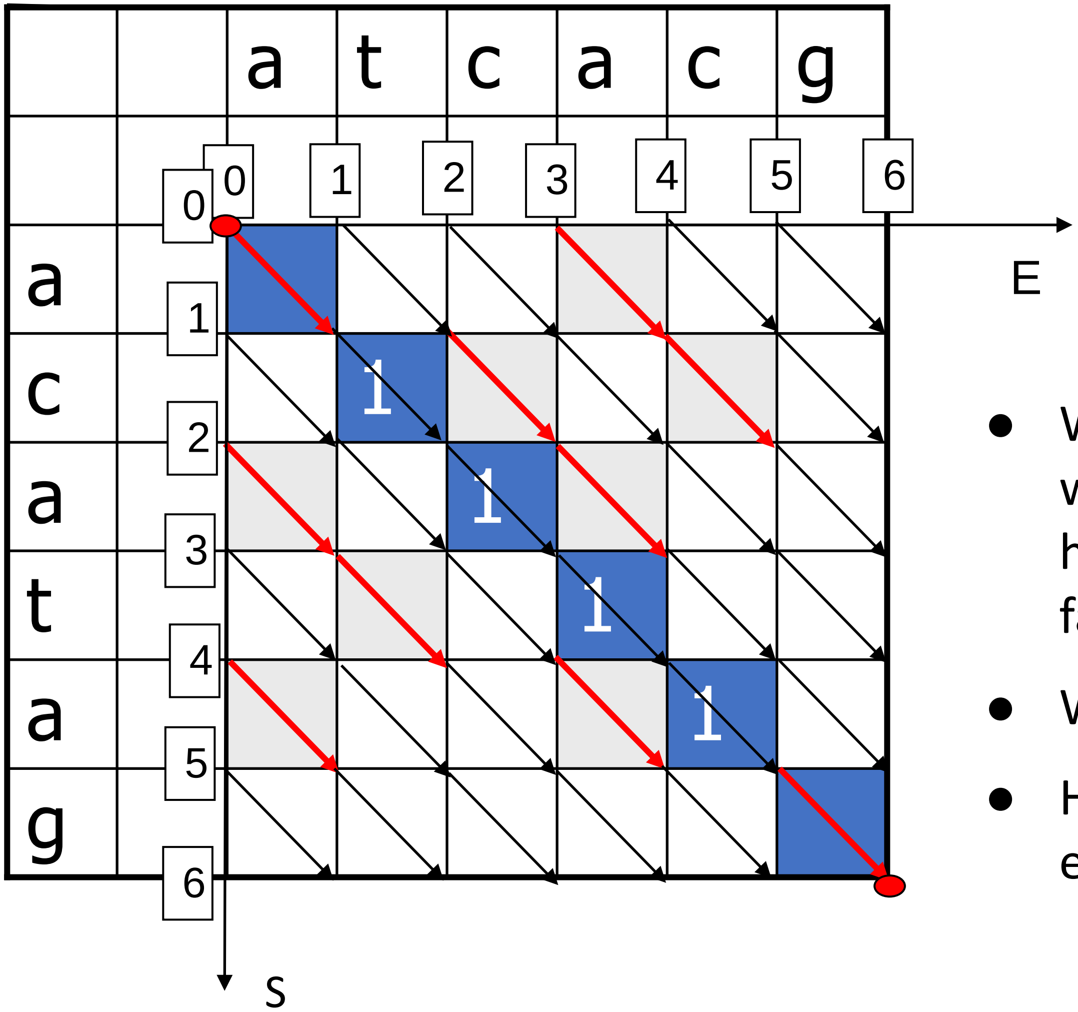


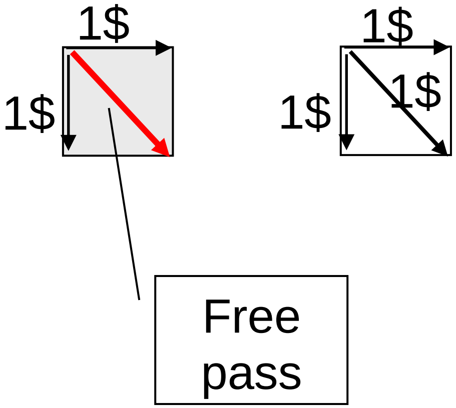
Output:

the cheapest path from (0,0) to (6,6)

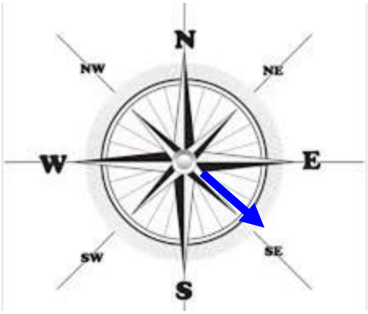
Without the map



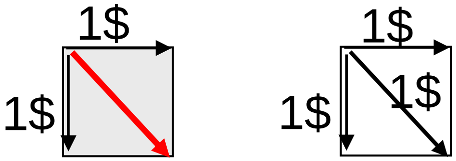


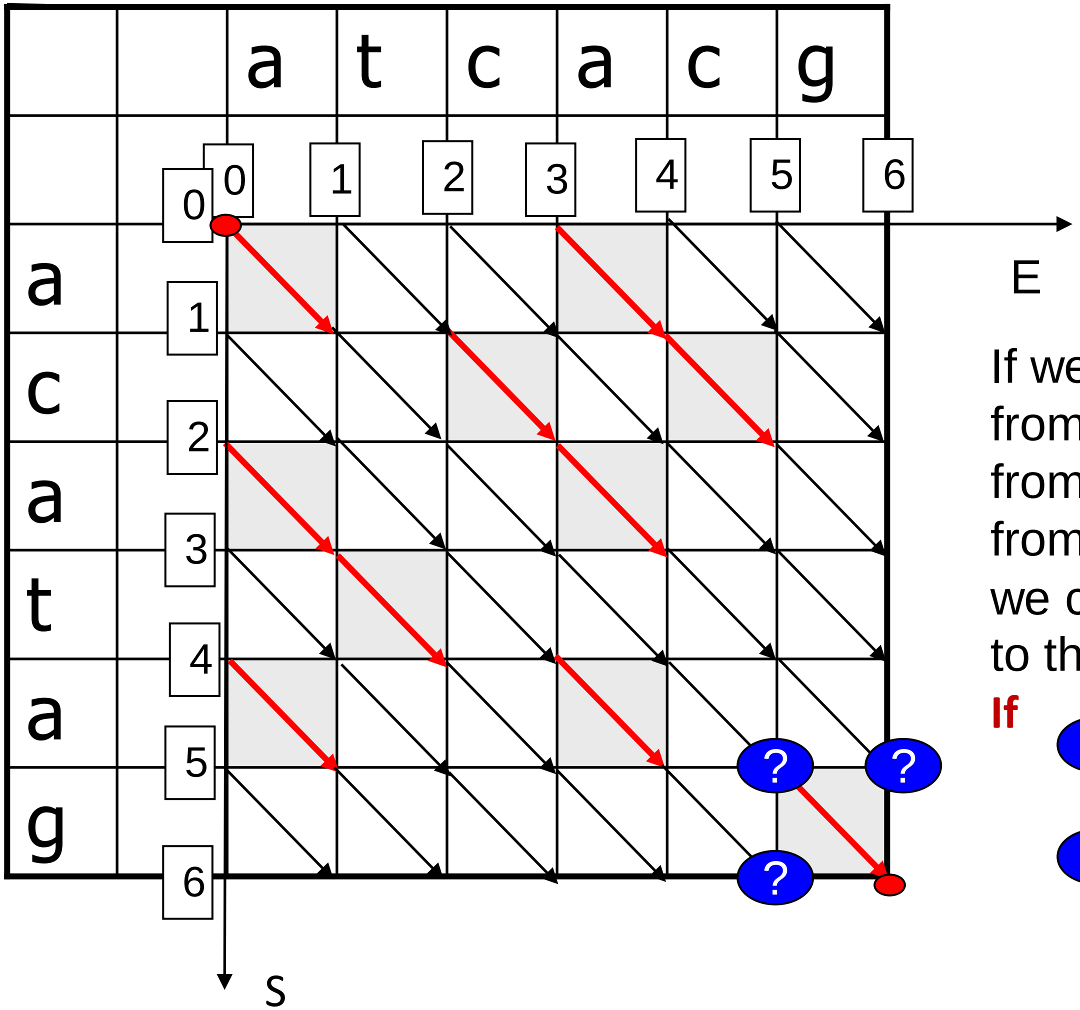


- Without additional information, we will always head South-East hoping to reach the destination faster
- We will pay 4$
- However a better (cheaper) path exists with more free cells



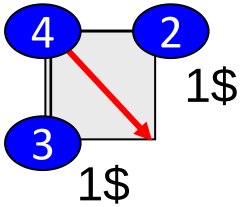
Sub-problems approach

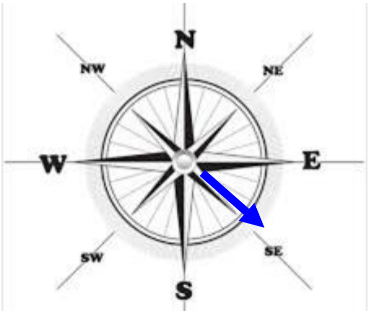




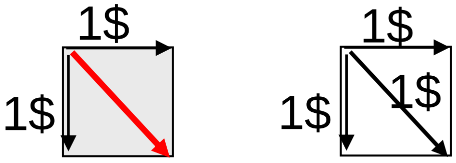
If we knew the cheapest paths
from (0,0) to (5,5)
from (0,0) to (6,5)
from (0,0) to (5,6)
we could choose the best last step
to the destination:

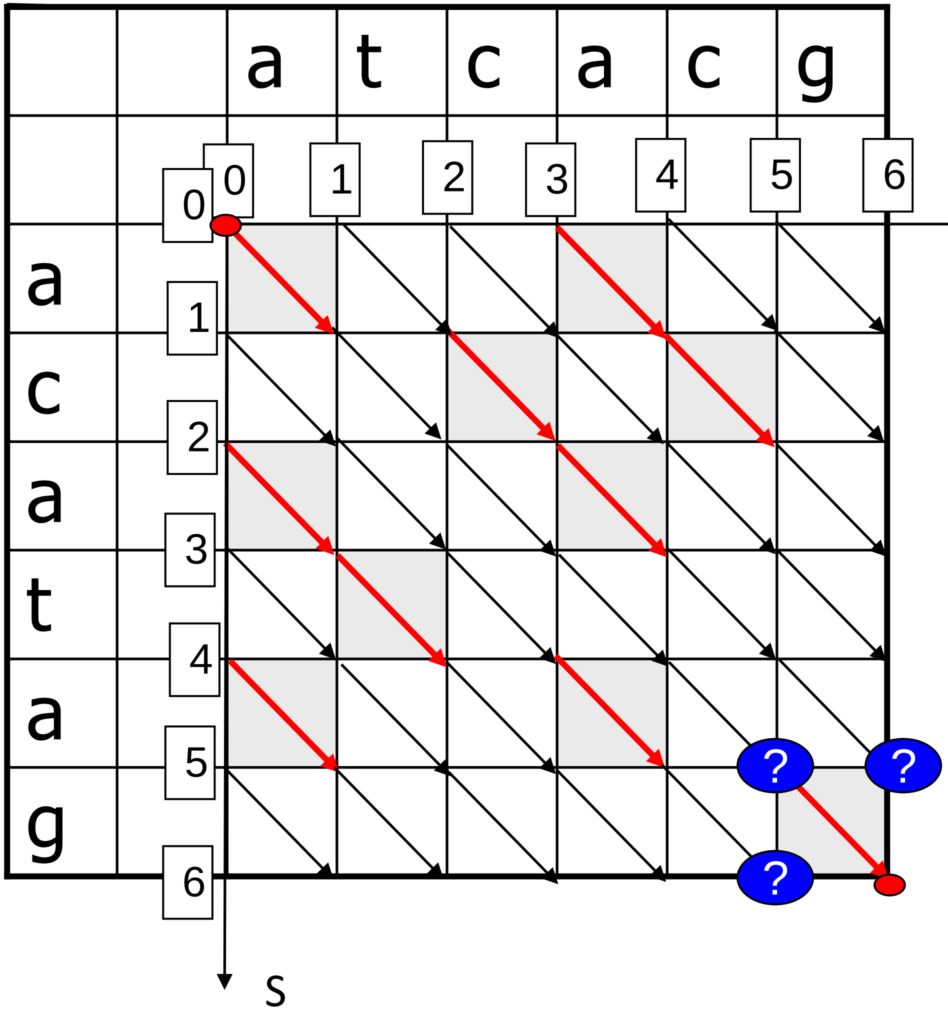
If





Sub-problems approach



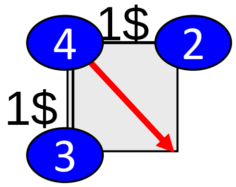


E

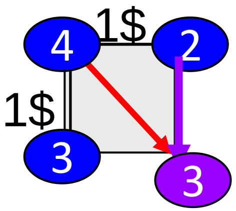
And this is true for any cell – what path to choose depends on the cheapest paths to the left, upper, and upper-left corner.

Since we choosing only 1 step, we can take the min of the result

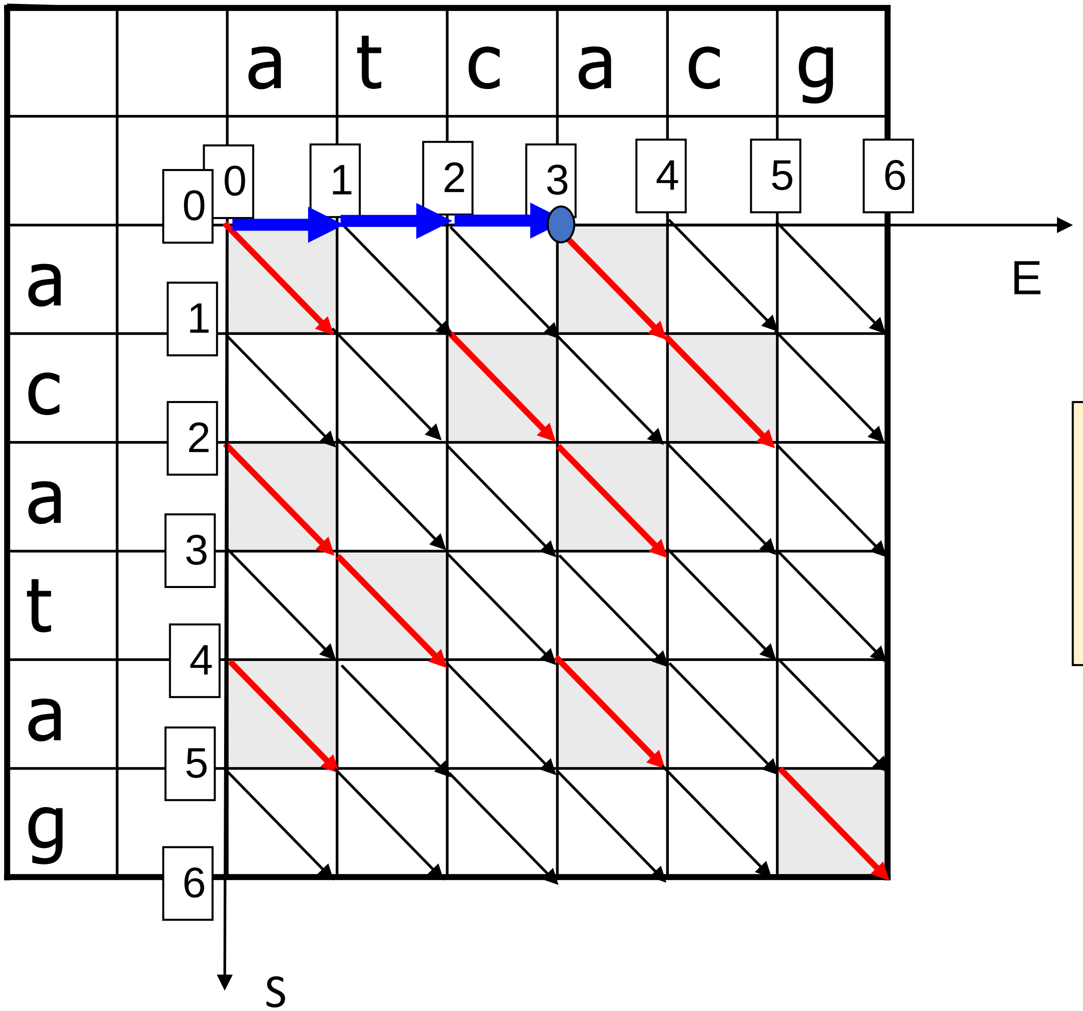
If



then



Recurrence relation – base condition



When i=0, there is no cheaper way of going from (0,0) to (0,j) than to pay j$ - heading strictly to the right (East)

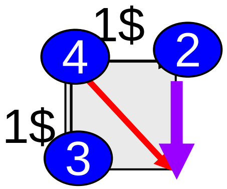
The same for j=0.

The base condition:

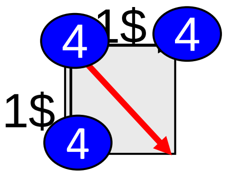
if i=0 then COST(i,j)=j

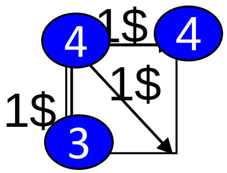
if j=0 then COST(i,j)=i

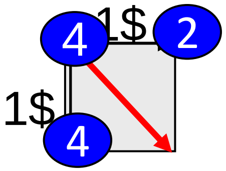
Recurrence relation (for i>0 and j>0)



$$COST(i,j) = \min \begin{cases} COST(i-1,j)+1 \\ COST(i,j-1)+1 \\ COST(i-1,j-1)+DIAGONAL(i,j) \end{cases}$$

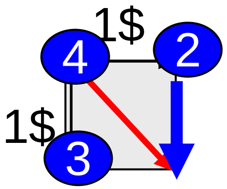






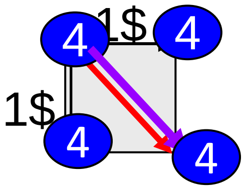
For each case, what is the best choice?

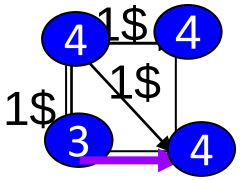
Recurrence relation (for i>0 and j>0)

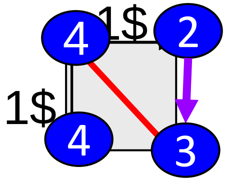


$$\text{COST(i,j)= min}\begin{cases}\text{COST(i-1,j)+1}\\ \text{COST(i,j-1)+1}\\ \text{COST(i-1,j-1)+DIAGONAL(i,j)}\end{cases}$$

For each case, what is **the best choice**?







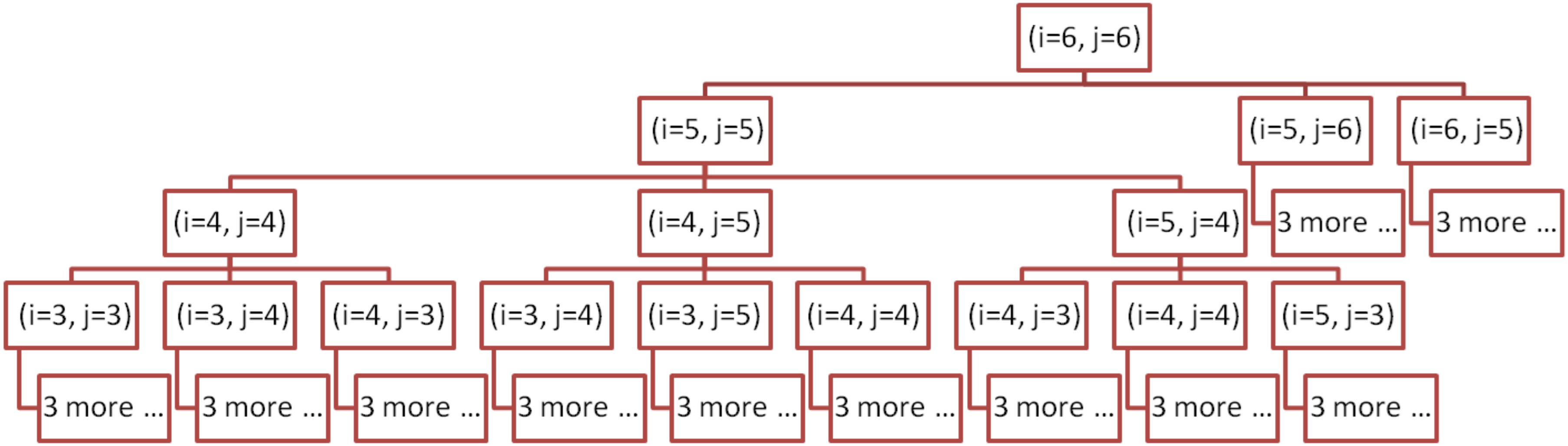
Recursive algorithm

$$\text{COST(i,j)= min} \begin{cases} \text{COST(i-1,j)+1} \\ \text{COST(i,j-1)+1} \\ \text{COST(i-1,j-1)+DIAGONAL(i,j)} \end{cases}$$

```
algorithm cheapestPath ( array diagonalCost, N, M )
        return cost ( N, M, diagonalCost )


algorithm cost ( i, j, diagonalCost)
        if i=0 then
                return j
        if j=0 then
                return i
        return min (cost ( i-1, j ) +1, cost ( i, j-1)+1, cost ( i-1, j-1)+diagonalCost [i] [j] )
```

The recursion tree: $O(3^N)$



(i=6, j=6)

(i=5, j=5)

(i=5, j=6)

(i=6, j=5)

(i=4, j=4)

(i=4, j=5)

(i=5, j=4)

3 more …

3 more …

(i=3, j=3)

(i=3, j=4)

(i=4, j=3)

(i=3, j=4)

(i=3, j=5)

(i=4, j=4)

(i=4, j=3)

(i=4, j=4)

(i=5, j=3)

3 more …

3 more …

3 more …

3 more …

3 more …

3 more …

3 more …

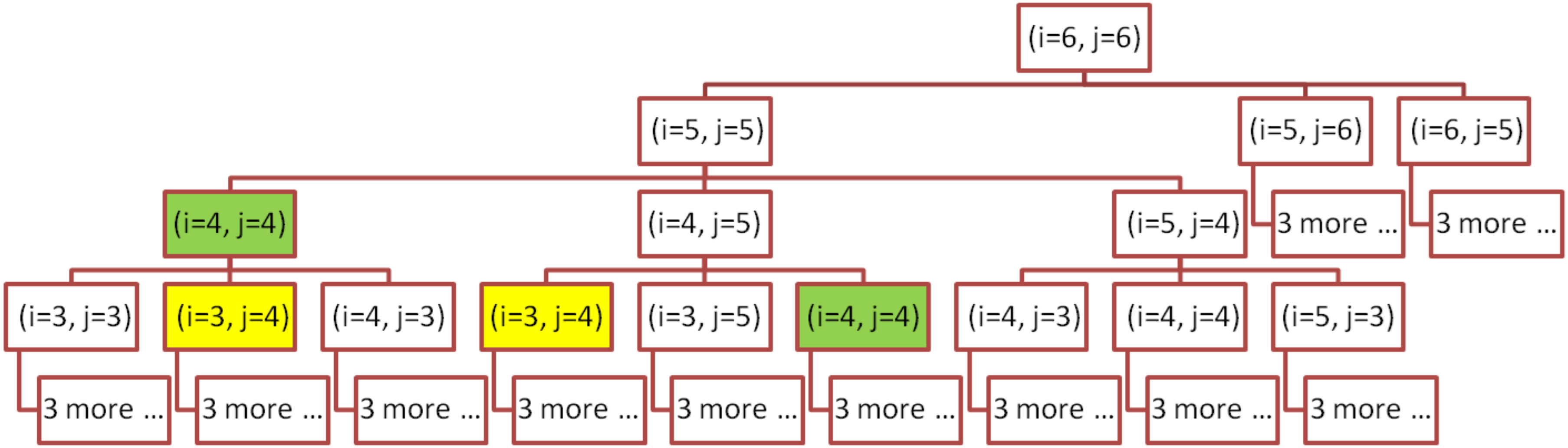
3 more …

3 more …

$O(3^N)$?

But there are only N*M different combinations (*i*,*j*)!

Recursive algorithm: O(3^N)



(i=6, j=6)

(i=5, j=5) (i=5, j=6) (i=6, j=5)

(i=4, j=4) (i=4, j=5) (i=5, j=4) 3 more ... 3 more ...

(i=3, j=3) (i=3, j=4) (i=4, j=3) (i=3, j=4) (i=3, j=5) (i=4, j=4) (i=4, j=3) (i=4, j=4) (i=5, j=3)

3 more ... 3 more ... 3 more ... 3 more ... 3 more ... 3 more ... 3 more ... 3 more ... 3 more ...

The algorithm is exponential in N because **we call the recursive function multiple times with the same parameters!**

Idea 1: store intermediate results

- Store the results of the *cost(i,j)* in a 2D table – so they do not need to be recomputed when needed again
- There are at most N^2 different combinations of (*i,j*)
- For each combination of (i,j) we compute the cost(i,j) only once
- When we need cost(i,j) again, we first check if it is already computed
- This gives a total running time $O(N^2)$

- The method of storing the results of recursive calls in a lookup table is called ***recursion with memoization***

Idea 2: The bottom-up computation

- In this problem we would need to compute the cost for all combinations of (*i, j*)
- Instead of starting from cost(N,M) - fill in the best values for each cell of N*M table **starting from the lowest values**

The bottom-up computation

- Create a table of size(*N*x*M*) to store results of cost(*i, j*) for each $0 \leq i \leq N$ and $0 \leq j \leq M$
- First, fill-in the basic values of recursion – for *i*=0 and for *j*=0
- Apply recursive formula for computing the value of each cell from the lowest numbers of *i* and *j* to the highest (by rows or by columns)

- At the end, we will have the cost of the best path in the cell (*N, M*)

The recurrence relation: stays the same

The base condition:

if i=0 then COST(i,j)=j
if j=0 then COST(i,j)=i

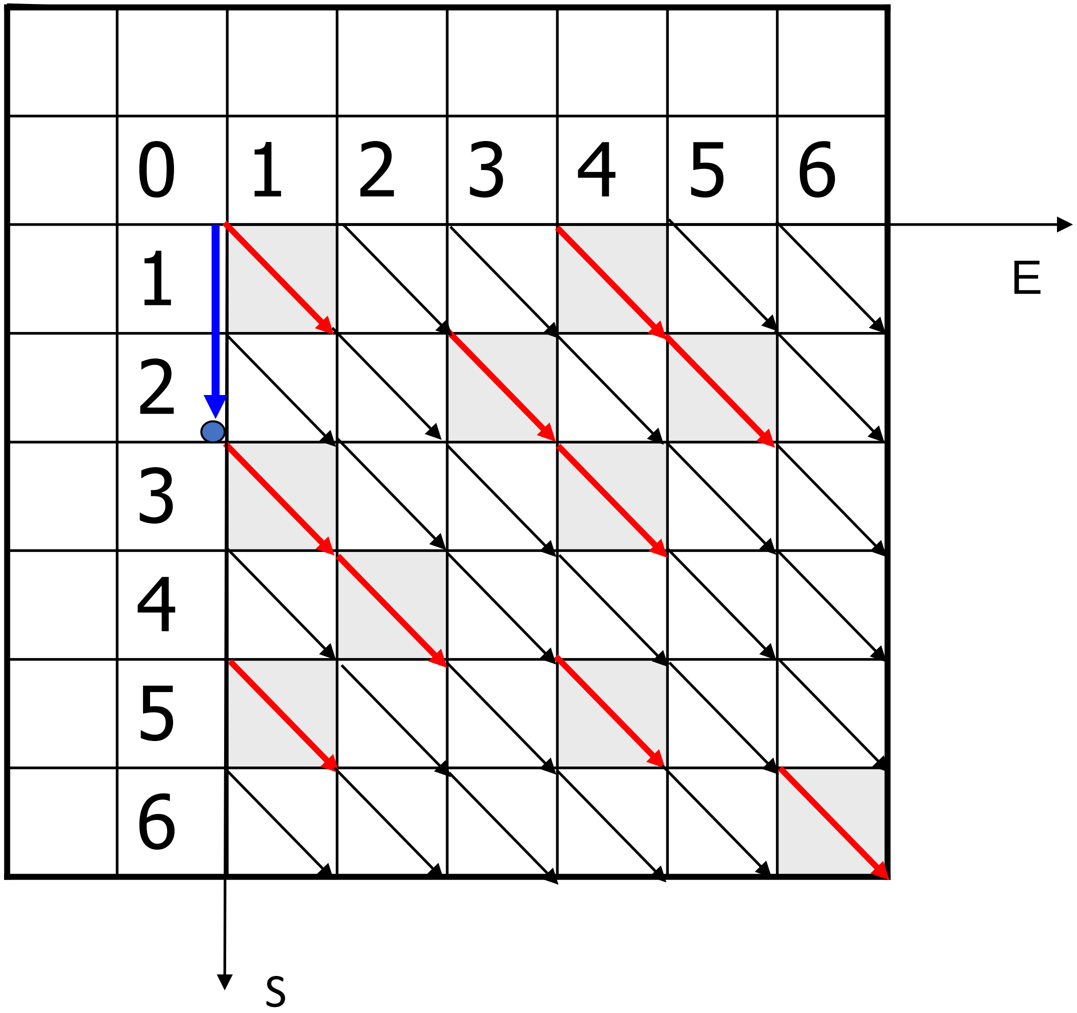
The main relation (for i>0 and j>0)

$$\mathrm{COST}(i,j)=\min\begin{cases}\mathrm{COST}(i-1,j)+1\\ \mathrm{COST}(i,j-1)+1\\ \mathrm{COST}(i-1,j-1)+\mathrm{DIAGONAL}(i,j)\end{cases}$$

We change:

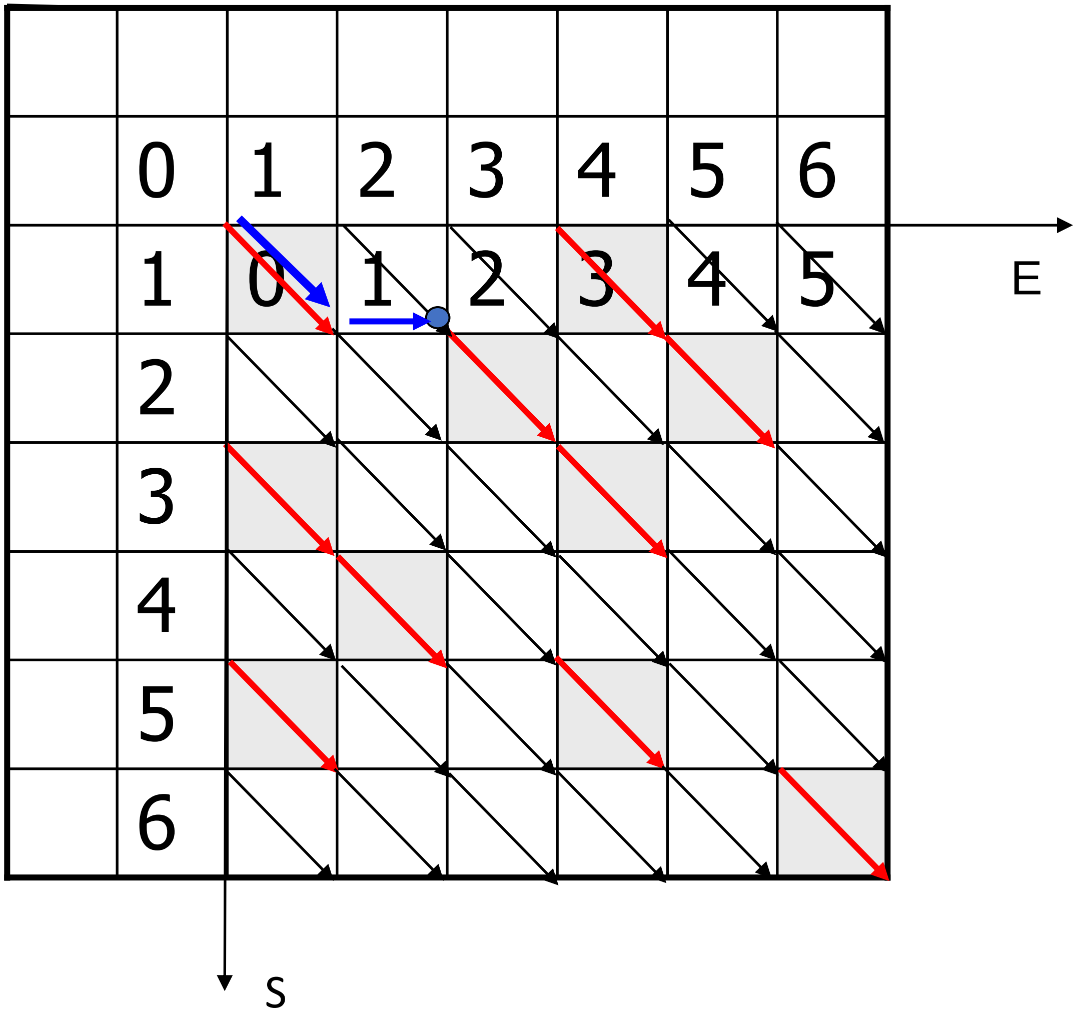
the order of computation

Fill values for i=0 and for j=0 (the base recursion condition)



There is no cheaper way of going to the point (2,0) than paying 2 $

Fill values for i=1 (from left to right)

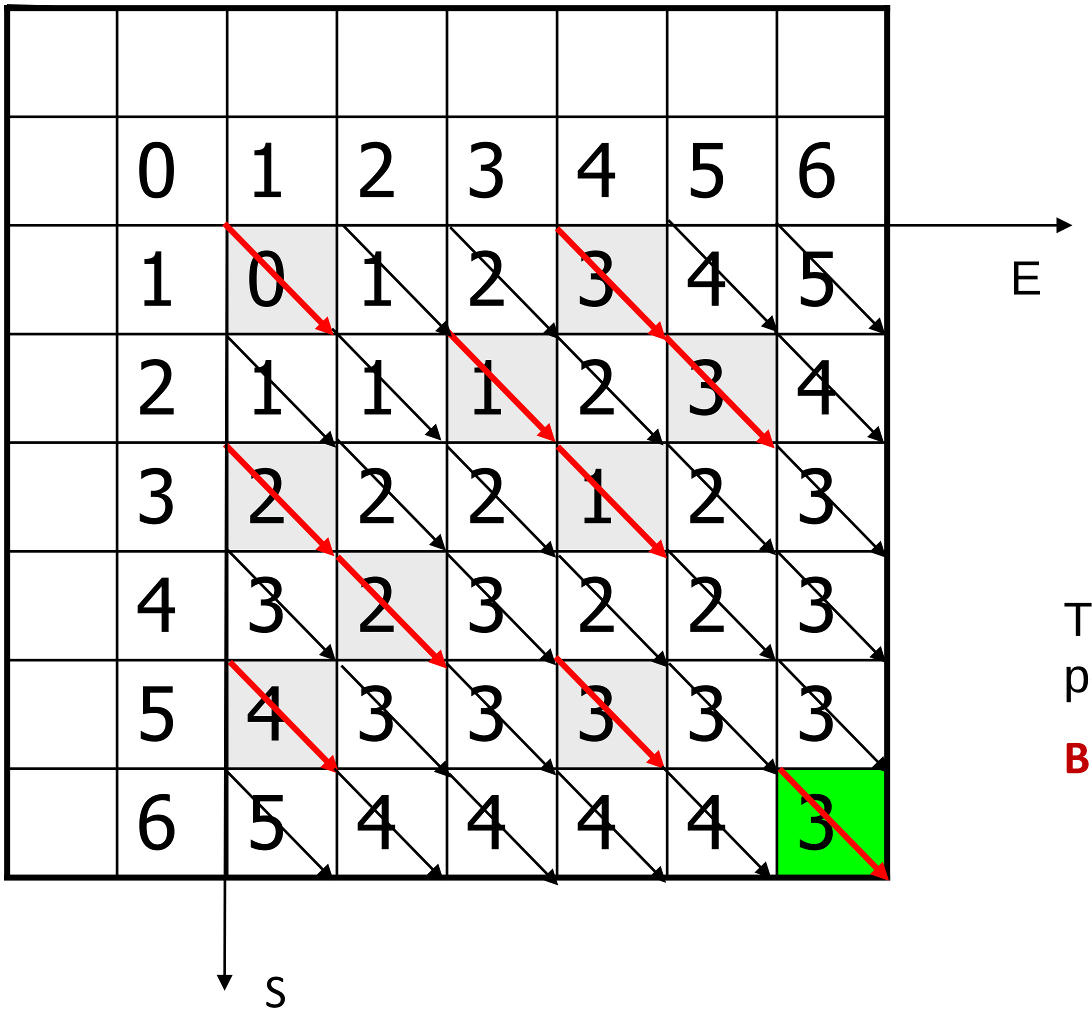


	0	1	2	3	4	5	6
	1	0	1	2	3	4	5
	2						
	3						
	4						
	5						
	6						

Cell(1,2)=1

since the cheapest possible way is to continue the free path through the cell (1,1)

Fill the entire table (left-to-right top-down)

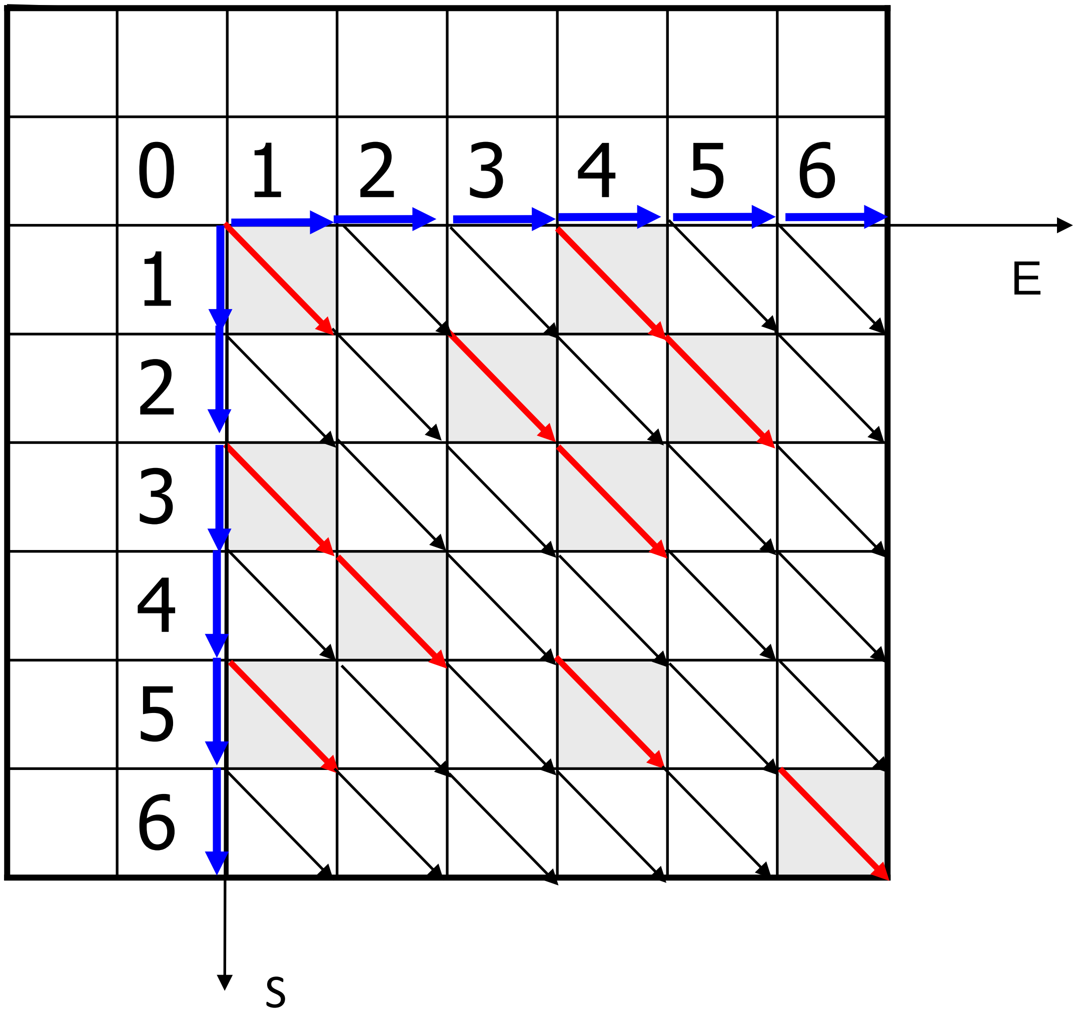


	0	1	2	3	4	5	6
	1	0	1	2	3	4	5
	2	1	1	1	2	3	4
	3	2	2	2	1	2	3
	4	3	2	3	2	2	3
	5	4	3	3	3	3	3
	6	5	4	4	4	4	3

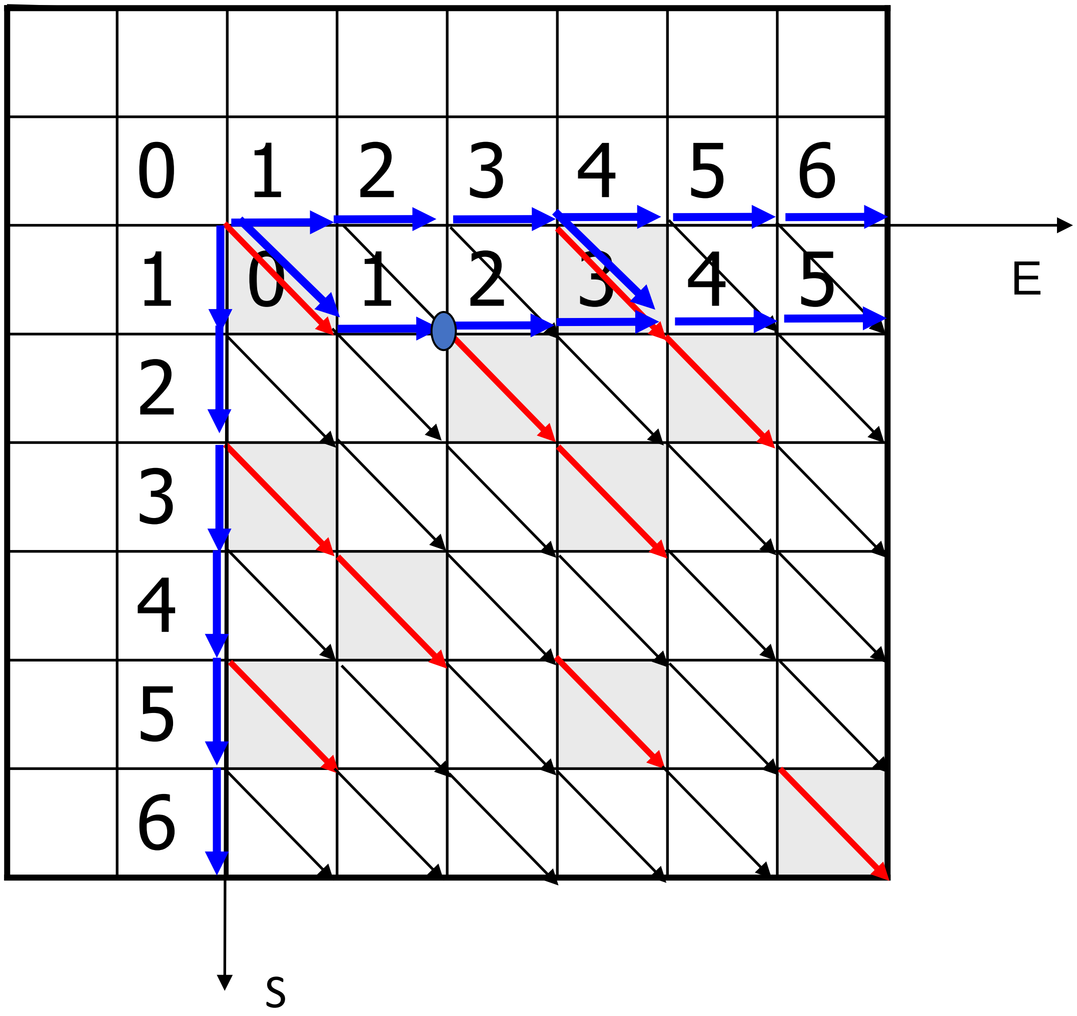
The overall cheapest possible path costs 3$

But what is this path?

Keeping track of the source

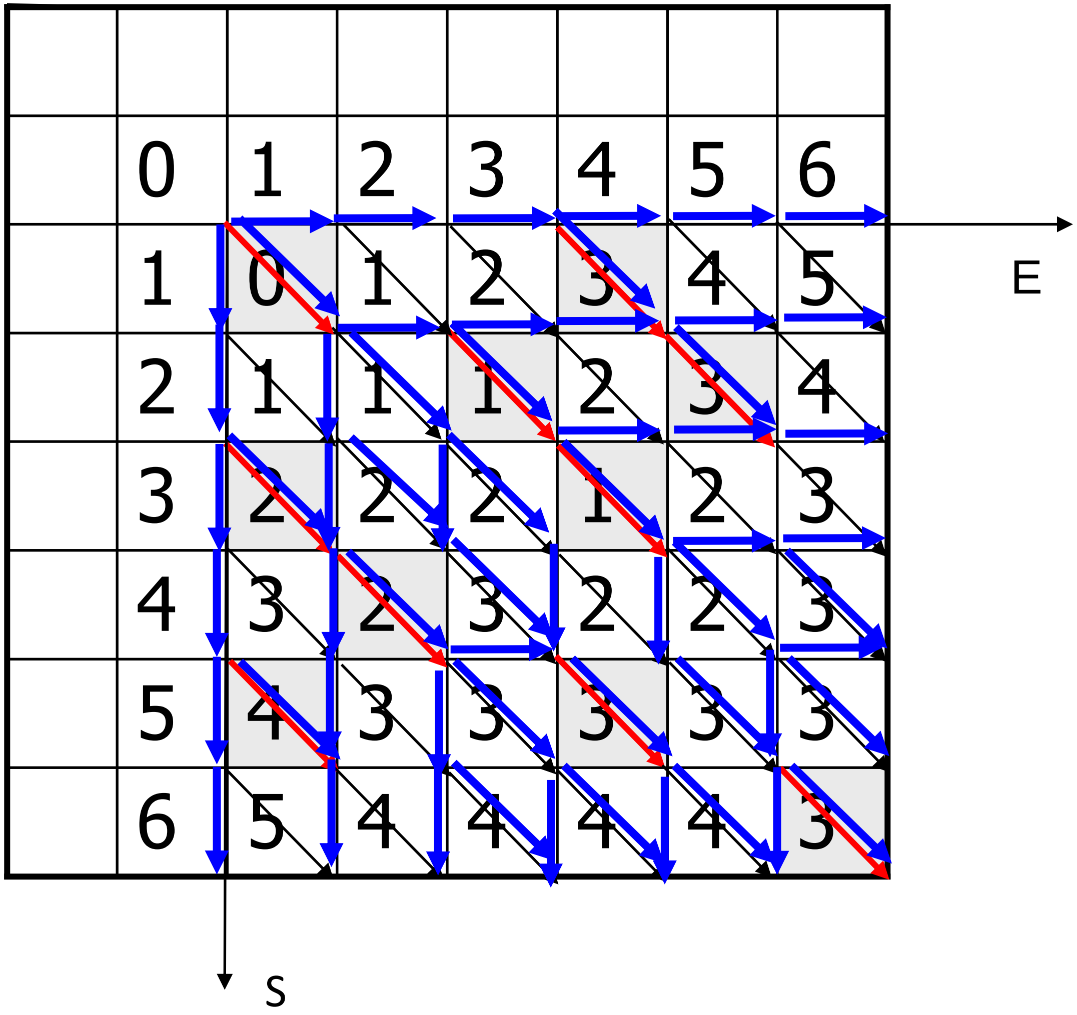


Keeping track of the source



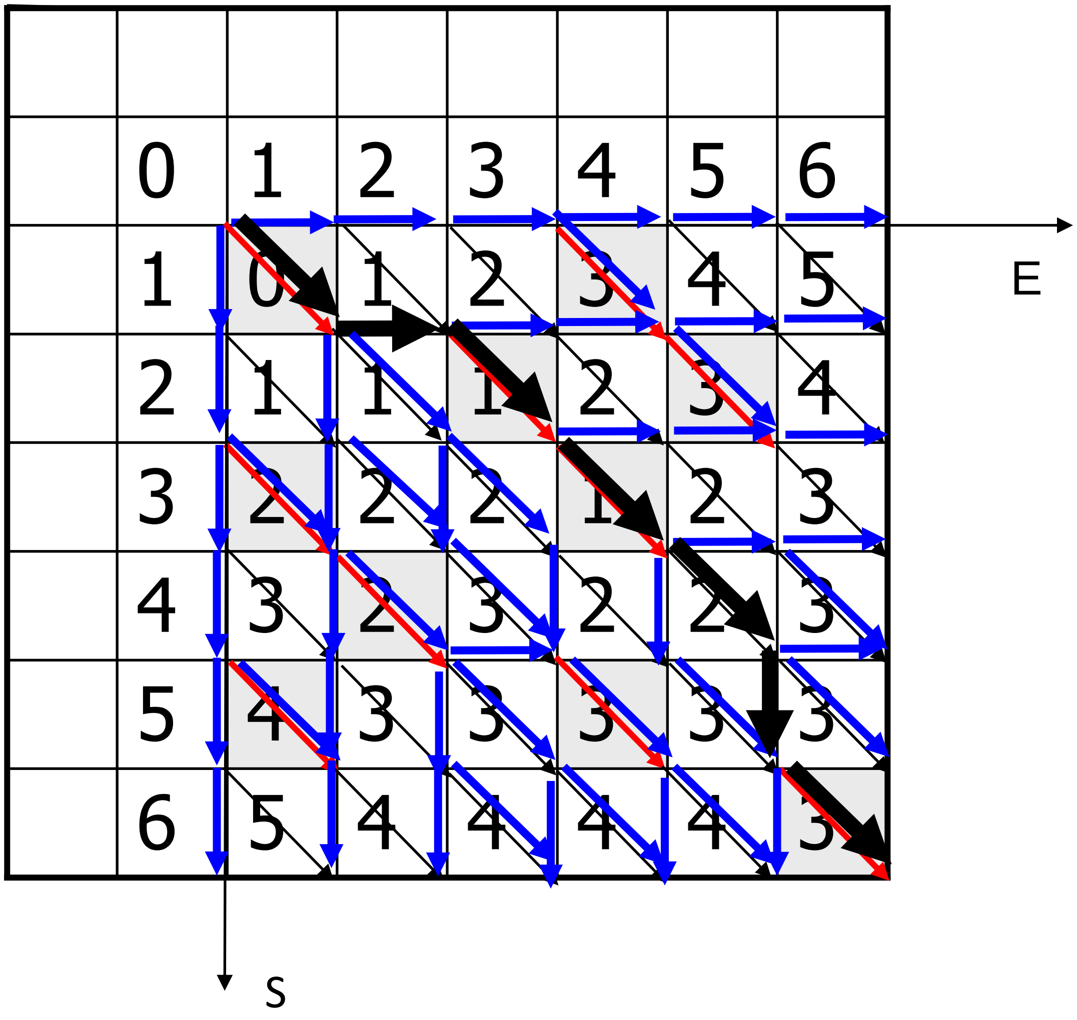
	0	1	2	3	4	5	6
	1	0	1	2	3	4	5
	2						
	3						
	4						
	5						
	6						

Keeping track of the source



	0	1	2	3	4	5	6
	1	0	1	2	3	4	5
	2	1	1	1	2	3	4
	3	2	2	2	1	2	3
	4	3	2	3	2	2	3
	5	4	3	3	3	3	3
	6	5	4	4	4	4	3

Trace back –
how did we get the path with the cost 3?



	0	1	2	3	4	5	6
	1	0	1	2	3	4	5
	2	1	1	1	2	3	4
	3	2	2	2	1	2	3
	4	3	2	3	2	2	3
	5	4	3	3	3	3	3
	6	5	4	4	4	4	3

Our *Dynamic Programming* algorithm

Algorithm: cheapestPath (diagonalCost NxM)

```
allocate array DPTable (NxM)
DPTable [0][0]:=0

for i from 1 to N:
    DPTable [i][0]:=i
for j from 1 to M:
    DPTable [0][j]:=j

for i from 1 to N:
    for j from 1 to M:
        DPtable [i][j]:=min (DPtable [i-1][j-1]+ diagonalCost [i][j],
                             DPtable [i-1][j]+1, DPtable [i][j-1]+ 1)

return DPTable [N][M]
```

2 nested loops: $O(N^2)$

Dynamic programming: when

- We want to optimize something: min, max
- The solution to the problem depends on the solutions to subproblems
- We would need the solutions to all subproblems
- Subproblems **overlap**

Dynamic programming: how

- The recurrence relation
- The bottom-up computation
- The traceback

“Programming” in “Dynamic programming” has nothing to do with programming!

- Richard Bellman developed this idea in 1950s working on an Air Force project
- At that time, his approach seemed completely impractical
- He wanted to hide that he is really doing pure math from the Secretary of Defense

Richard Bellman

. . . What name could I choose? I was interested in planning but *planning* is not a good word for various reasons. I decided therefore to use the word “programming” and I wanted to get across the idea that this was dynamic. **It was something not even a Congressman could object to.** So I used it as an umbrella for my activities.

Edit distance

Transforming one sequence into another: *edit operations*

- We can transform the first string S1 into the second S2 by applying a sequence of edit operations on S1 :
  - Deleting 1 symbol
  - Inserting 1 symbol
  - Replacing 1 symbol

S1	a	c	t			a	t	g
S2	a		t	a	c	a		g

Delete c

Insert a

Insert c

Delete t

In total, 4 edit operations

String alignment

❑ An *alignment* of 2 strings is obtained by first inserting spaces (gaps), either into or at the end of both strings, and then placing 2 resulting strings one above the other, so that every character or space in either string is opposite a single character or space in the other string

Alignment

S1	a	c	t	-	-	a	t	g
S2	a	-	t	a	c	a	-	g

4 gaps,

no mismatches

Edit distance: definition

- The ***edit distance*** between two strings is defined as the **minimum** number of edit operations needed to transform one string into another

S1	a	c	t	a	t		g
S2	a		t	a	c	a	g

Delete c

Replace t

Insert a

In total, 3 edit operations

Optimal alignment

- ❑An optimal alignment is obtained from the calculation of the edit distance

Optimal Alignment

S1	a	c	t	a	t		g
S2	a		t	a	c	a	g

Delete c

Replace t

Insert a

Edit distance=3

Is this really the smallest number of edit operations?

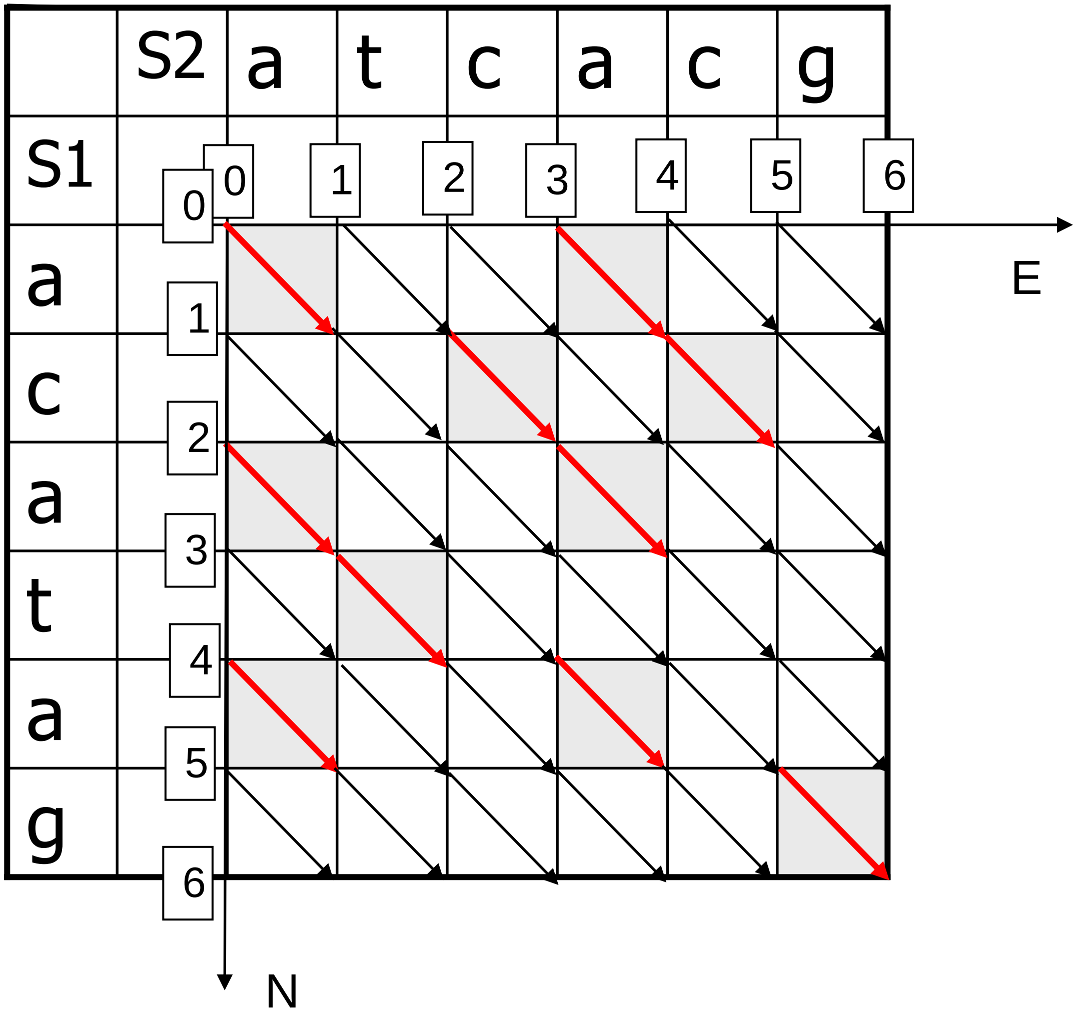
How do we compute edit distance in general?

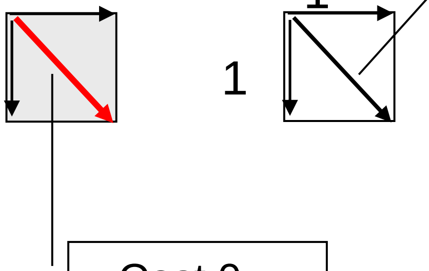
The edit distance problem

Input: 2 strings S_1 and S_2

Output: the *edit distance* between two strings along with a sequence of the operations which describe the transformation

Full analogy with the cheapest path





1 replacement

1 insertion

1 deletion

1

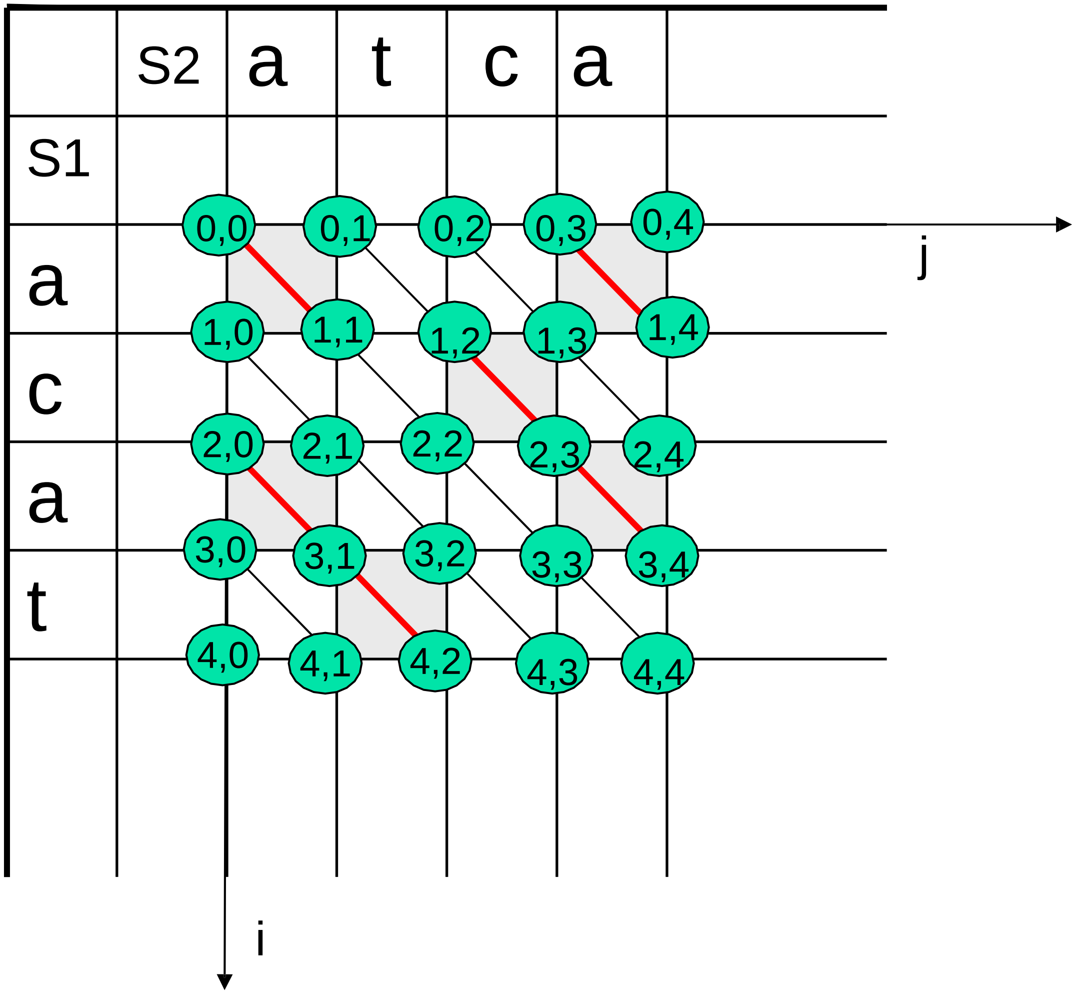
Cost 0 – characters match

The dynamic programming solution to the edit distance problem

Trivially follows from the solution for the cheapest path:

- If we moved strictly down in the grid, we inserted 1 symbol from S1
- If we moved strictly to the right, we deleted (ignored) 1 symbol of S1
- If we moved by diagonal of cost 0, we matched the corresponding characters
- If we moved by diagonal of cost 1, we replaced one symbol in S1 with the corresponding symbol in S2

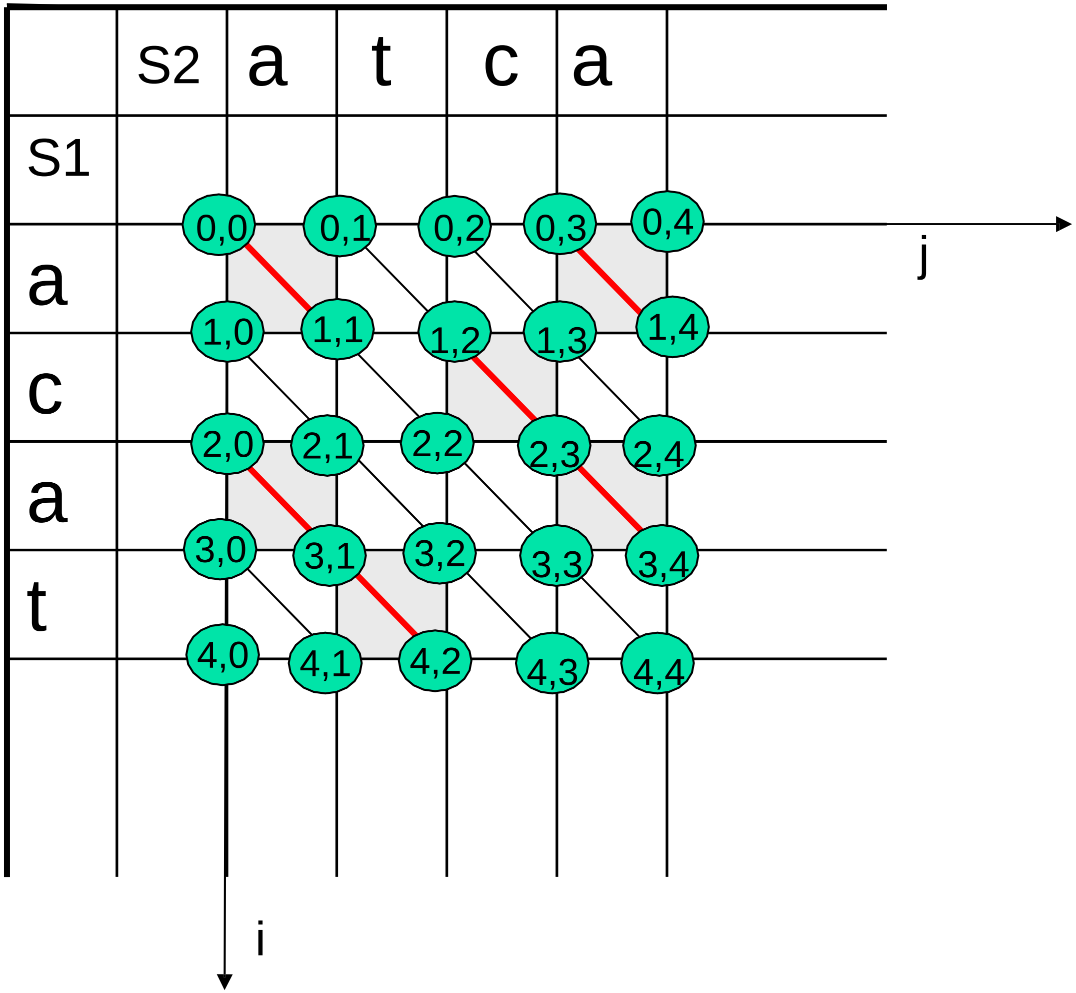
Useful abstraction: *edit graph*



An ***edit graph*** for a pair of strings S_1 and S_2 has $(N+1)*(M+1)$ vertices, each labeled with a corresponding pair (i,j), $0 \leq i \leq N$, $0 \leq j \leq M$

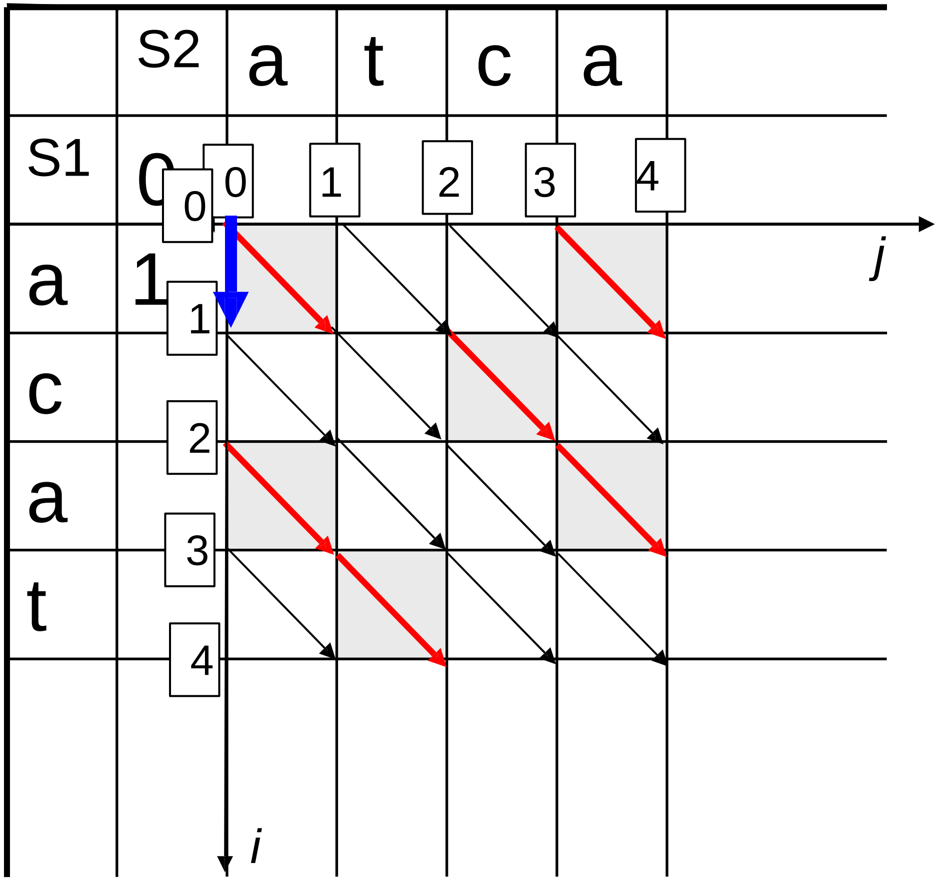
The edges are **directed** and their weight depends on the specific string problem: for the edit distance problem – red edges have cost 0, black edges have cost 1

The cheapest path in the edit graph



The cost of a **cheapest path** from vertex (0,0) to vertex (*N*,*M*) in this edit graph corresponds to the **edit distance** between S1 and S2, and the path itself represents a series of edit operations and an optimal alignment of S1 with S2

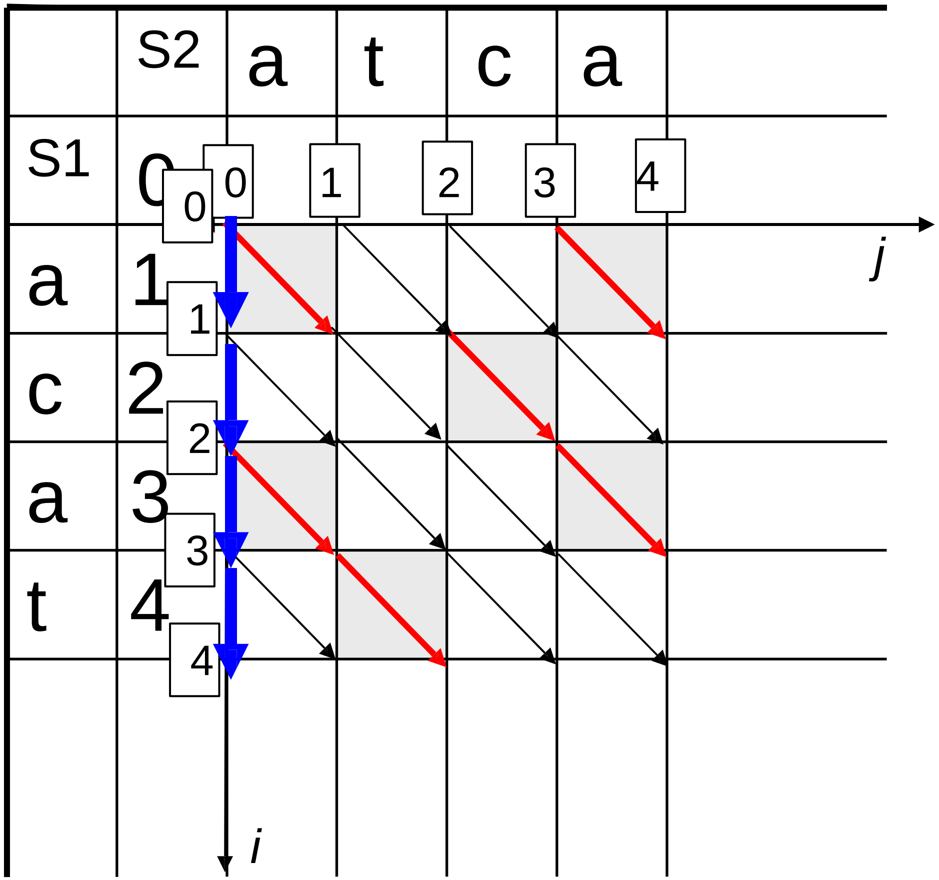
Calculating edit distance. Base condition



The minimum number of edit operations we need in order to transform string *a* into an empty string (of length 0) is 1 (deletion)

Therefore the minimum edit distance between ε and *a* is 1

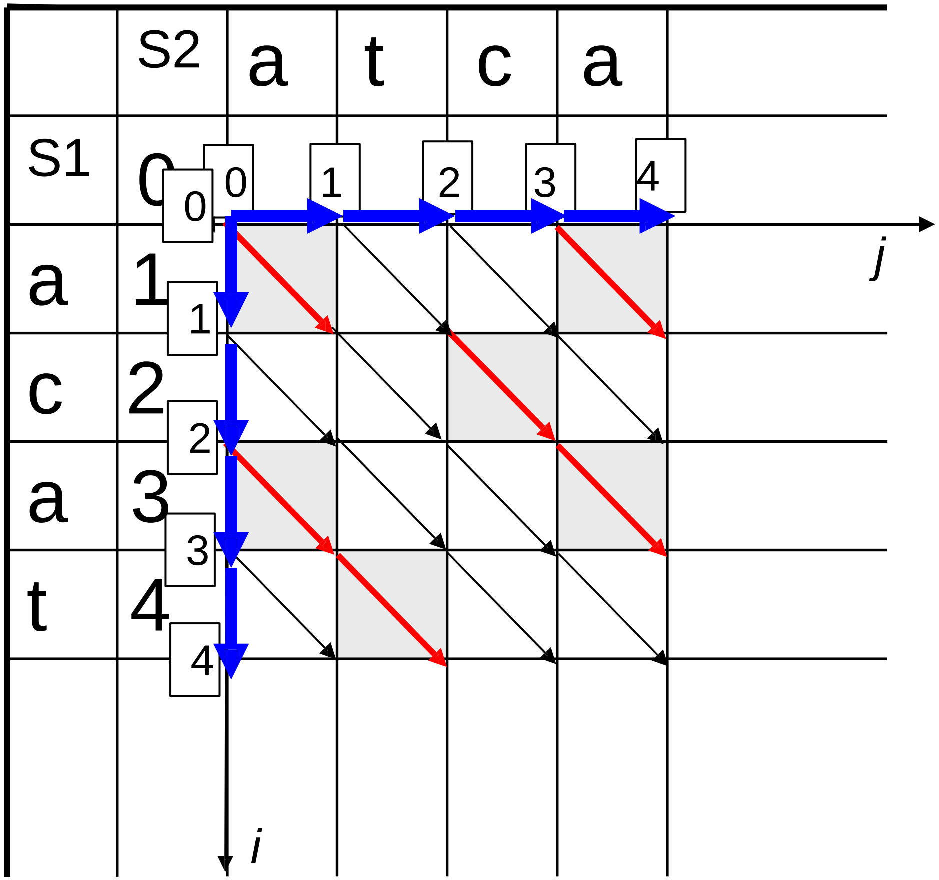
Calculating edit distance. Base condition



	S2	a	t	c	a
S1	0				
a	1				
c	2				
a	3				
t	4				

The same is true for ε and *ac*, *aca*, *acat*

Calculating edit distance. Base condition

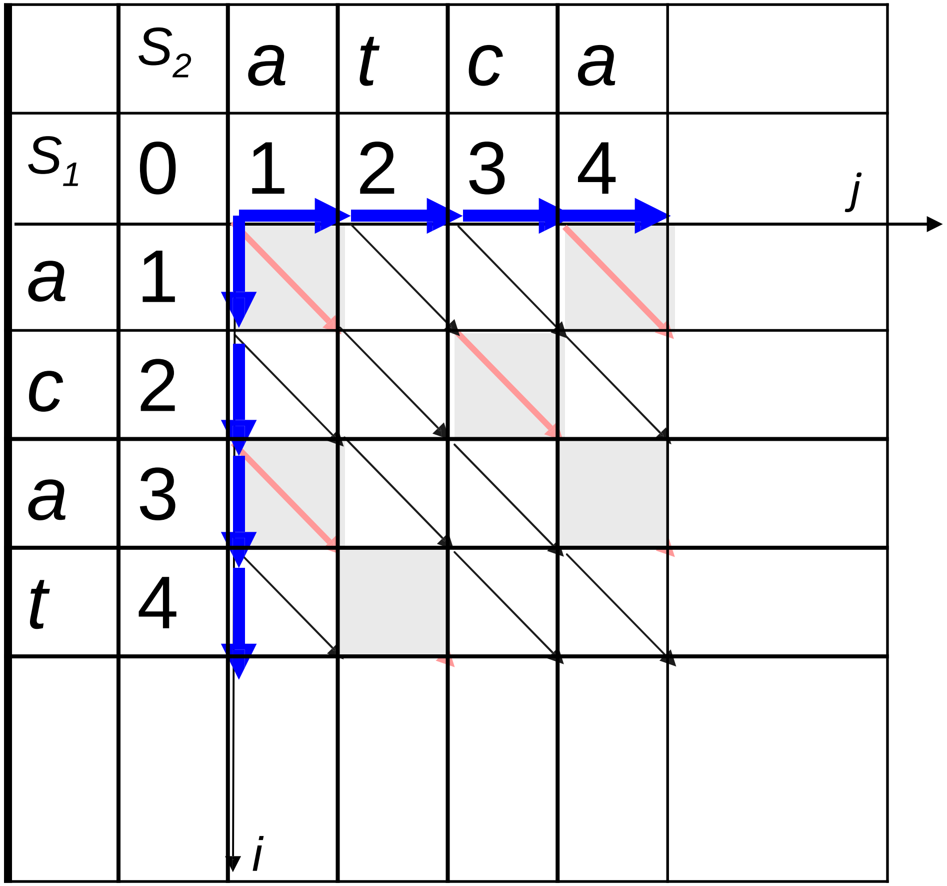


	S2	a	t	c	a
S1	0				
a	1				
c	2				
a	3				
t	4				

In order to transform ε into *a*, we need to insert 1 character. This is the best way to do it, there is no cheaper way.

The same for transforming ε into *at, atc, atca* with 2, 3, 4 insertions respectively

Calculating edit distance. Filling cells for i>0 and j>0



	S_2	a	t	c	a	
S_1	0	1	2	3	4	
a	1					
c	2					
a	3					
t	4					

There are only 3 different ways to move through the next cell in the graph:

1. Increase both *i* and *j* (diagonal)

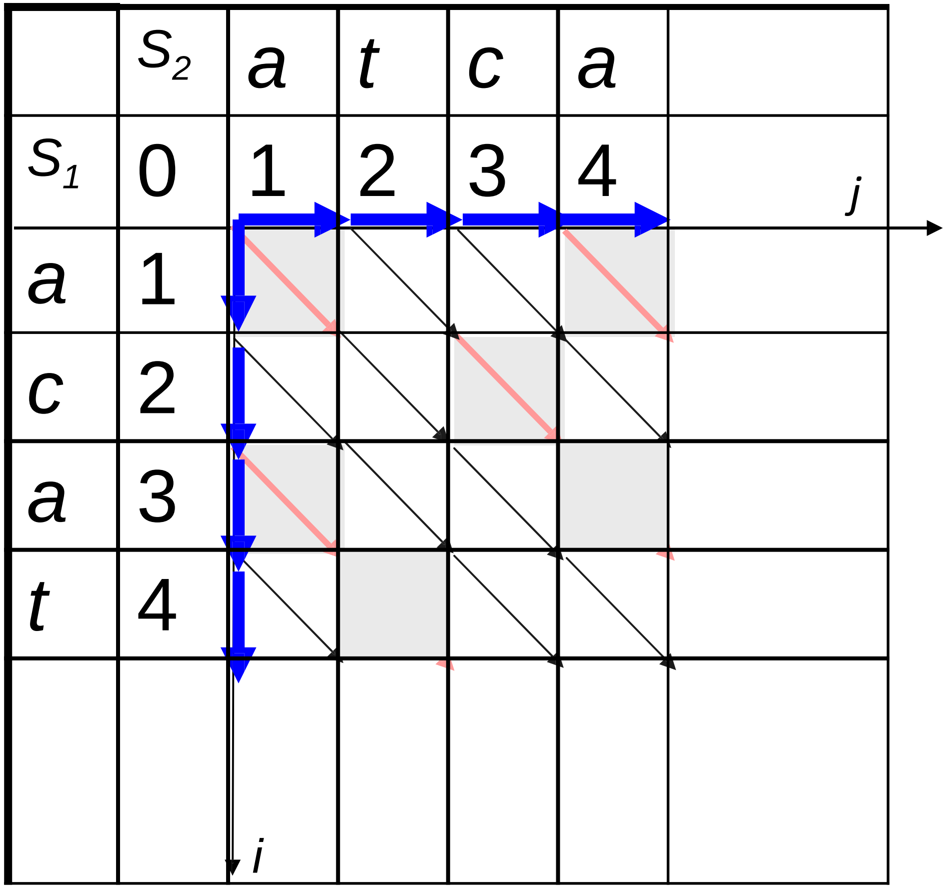
   if S1[i]≠S2[j] : 1 edit

   if S1[i]=S2[j] : 0 edits

1. Increase only *i* (insert $S_1[i]$) with the cost 1

2. Increase only *j* (delete - ignore $S_1[i]$) with the cost 1

Calculating edit distance. Filling cells for i>0 and j>0



	S_2	*a*	*t*	*c*	*a*	
S_1	0	1	2	3	4	
a	1					
c	2					
a	3					
t	4					

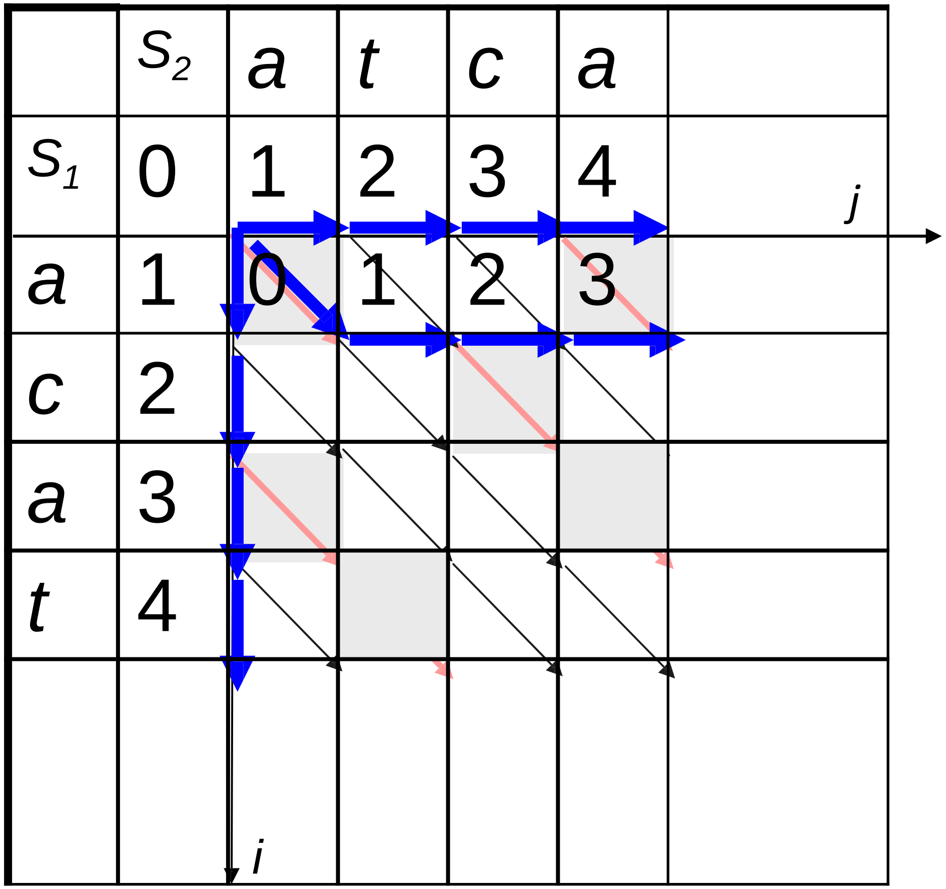
Thus, if we know the edit distance

D[*i*-1,*j*-1], *D*[*i*-1,*j*] and *D*[*i*, *j*-1], we can correctly calculate *D*[*i*,*j*]

This is true since there are no other ways of moving through cell [*i*][*j*].

Reaching the top, left and top-left corners by different paths cannot produce a better value than is already in these 3 cells, since they contain the minimum cost by definition

Calculating edit distance.
Filling cells for i>0 and j>0

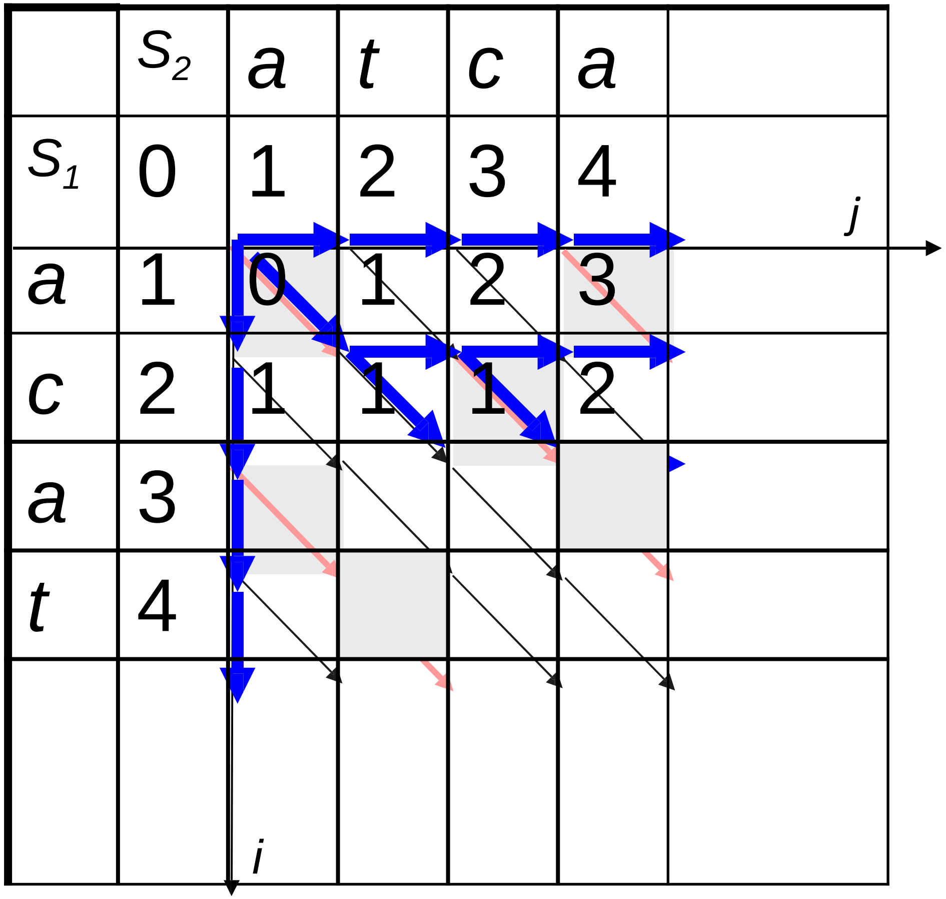


	S_2	*a*	*t*	*c*	*a*	
S_1	0	1	2	3	4	
a	1	0	1	2	3	
c	2					
a	3					
t	4					

$$D(i,j) \;=\; \min \begin{cases} D(i-1,j)+1 \\ D(i,j-1)+1 \\ D(i-1,j-1)+c(i,j) \end{cases}$$

$$\text{where } c(i,j) \;=\; \begin{cases} 0 \text{ if } S1[i]=S2[j] \\ 1 \text{ if } S1[i] \neq S2[j] \end{cases}$$

Calculating edit distance. Filling cells for i>0 and j>0



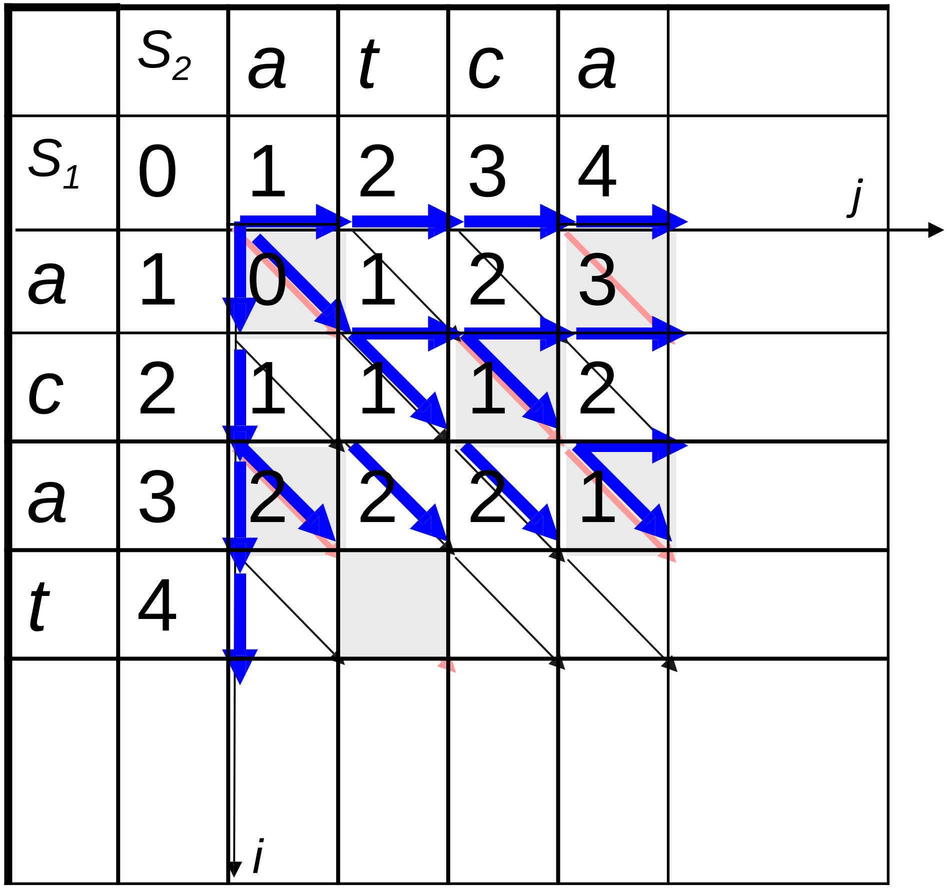
	S_2	*a*	*t*	*c*	*a*	
S_1	0	1	2	3	4	
a	1	0	1	2	3	
c	2	1	1	1	2	
a	3					
t	4					

$$D(i,j) = \min \begin{cases} D(i-1,j)+1 \\ D(i,j-1)+1 \\ D(i-1,j-1)+c(i,j) \end{cases}$$

$$\text{where } c(i,j) = \begin{cases} 0 \text{ if } S1[i]=S2[j] \\ 1 \text{ if } S1[i] \neq S2[j] \end{cases}$$

Calculating edit distance.

Filling cells for i>0 and j>0

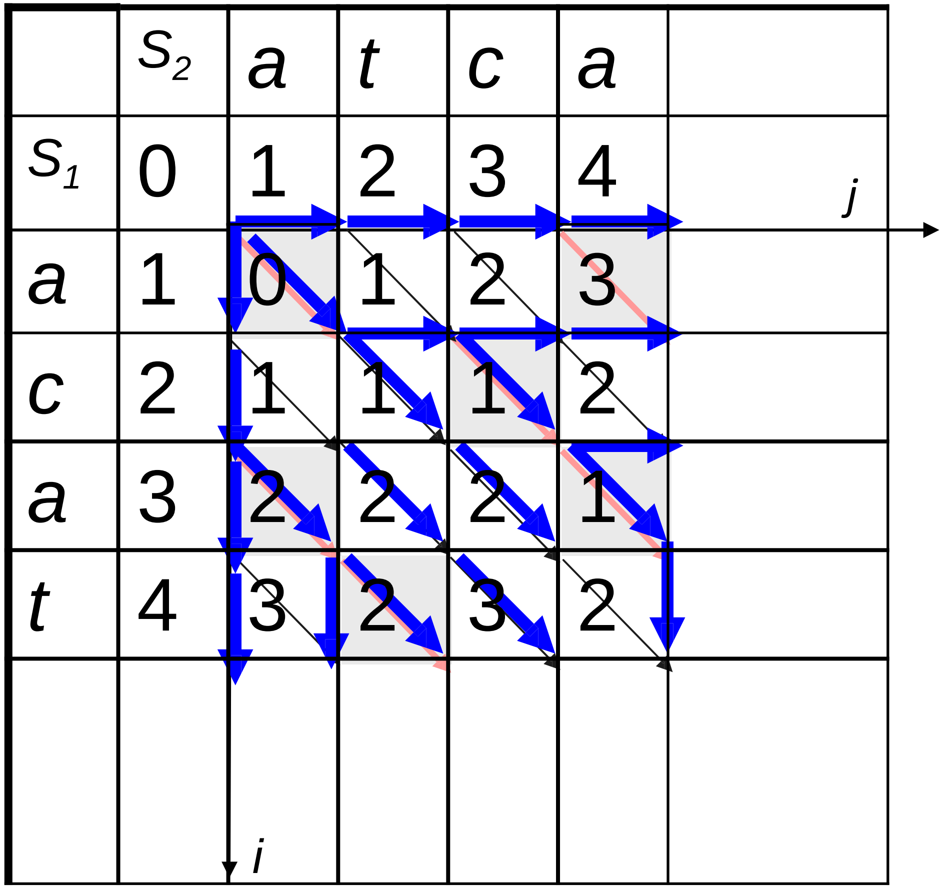


	S_2	*a*	*t*	*c*	*a*	
S_1	0	1	2	3	4	
a	1	0	1	2	3	
c	2	1	1	1	2	
a	3	2	2	2	1	
t	4					

$$D(i,j \quad = \quad \min \begin{cases} D(i-1,j)+1 \\ D(i,j-1)+1 \\ D(i-1,j-1)+c(i,j) \end{cases}$$

$$\text{where } c(\mathrm{i,j}) \; = \begin{cases} 0 \text{ if } S1[i]=S2[j] \\ 1 \text{ if } S1[i] \neq S2[j] \end{cases}$$

Calculating edit distance.
Filling cells for i>0 and j>0

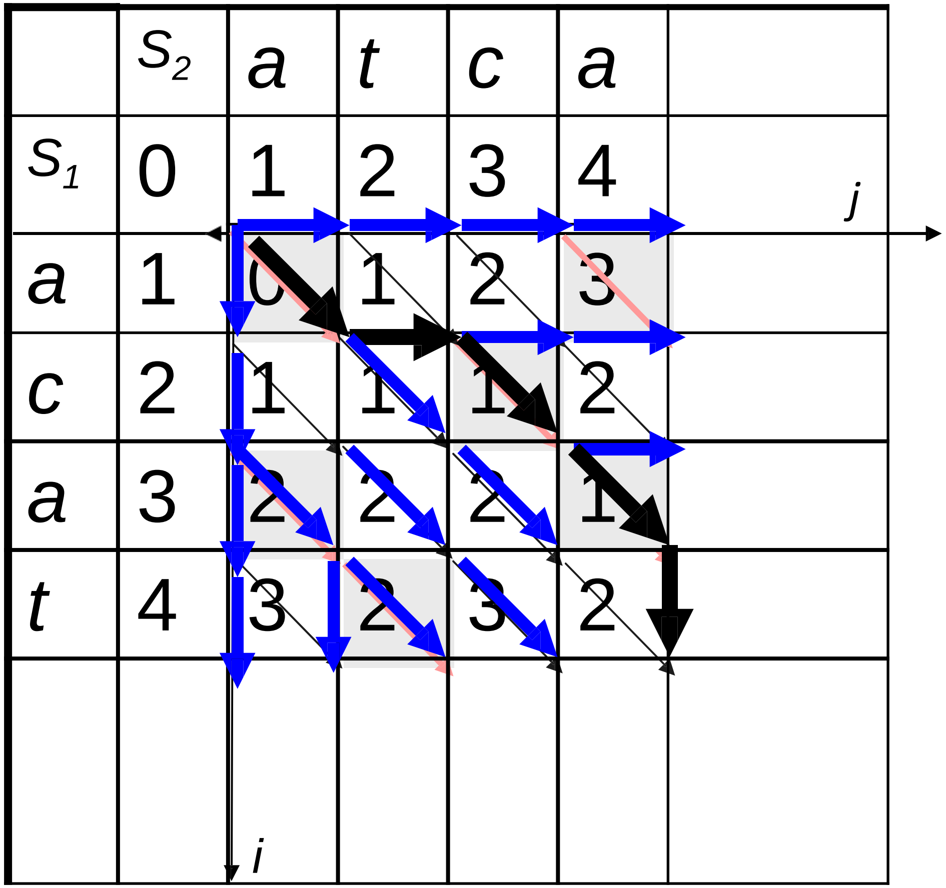


	S_2	*a*	*t*	*c*	*a*	
S_1	0	1	2	3	4	
a	1	0	1	2	3	
c	2	1	1	1	2	
a	3	2	2	2	1	
t	4	3	2	3	2	

$$D(i,j \;=\; \min \begin{cases} D(i-1,j)+1 \\ D(i,j-1)+1 \\ D(i-1,j-1)+c(i,j) \end{cases}$$

$$\text{where } c(\mathrm{i,j}) \;=\; \begin{cases} 0 \text{ if } S1[i]=S2[j] \\ 1 \text{ if } S1[i] \neq S2[j] \end{cases}$$

The sequence of edit operations



	S_2	*a*	*t*	*c*	*a*	
S_1	0	1	2	3	4	
a	1	0	1	2	3	
c	2	1	1	1	2	
a	3	2	2	2	1	
t	4	3	2	3	2	

↘ Place a character in S1 opposite to a character in S2

↓ Place a character in S1 opposite to a gap in S2

→ Place a character in S2 opposite to a gap in S1

S1	*a*	-	*c*	*a*	*t*
S2	*a*	*t*	*c*	*a*	-

Optimal alignment

S1	*a*	-	*c*	*a*	*t*
S2	*a*	*t*	*c*	*a*	-

Explanation:

S_2 can be obtained from S_1 by a series of the following edit operations:

Insertion of *t* at position 2

Deletion of *t* at position 5

An optimal alignment is not unique

S1	*-*	*a*	*t*	*t*	*a*	*a*	*g*
S2	*t*	*a*	*-*	*t*	*c*	*a*	*g*

S1	*-*	*a*	*t*	*t*	*a*	*a*	*g*
S2	*t*	*a*	*t*	*c*	*a*	*-*	*g*

2 different alignments with the optimal edit distance 3