Combination of **Dynamic Programming** and **Divide-and-Conquer**

Edit Distance in linear space

Algorithm by Hirschberg

Lecture 07.05

by Marina Barsky

Complexity of Edit Distance computation

- Quadratic - O(*NM*),

Where *N* is the length of S_1,

M is the length of S_2

What if *N* and *M* are very large? 2 scalability problems:

- Quadratic running time
- **Quadratic space**

Complexity of Edit Distance computation

- The time complexity is proportional to the number of edges in the edit graph: O(NM)

- The space complexity is proportional to the number of vertices in the edit graph, since for each vertex (i,j) we need to store the best incoming edge (or the value of $ED_{i,j}$): O(NM)

ED computation. Space

- For very long strings, the quadratic computation time is not as bad as the quadratic space required to store all the traceback pointers

- The quadratic space is a bottleneck of these algorithms

If we only want the value D[N][M]

		a	t	c	a	t	g
	0	1	2	3	4	5	6
a	1	0	1	2	3	4	5
c							
a							
t							
a							
g							

If we only want the value D[N][M]

		a	t	c	a	t	g
a	1	0	1	2	3	4	5
c	2	1	1	1	2	3	4
a							
t							
a							
g							

We do not need row 0 for computing values in row 3

If we only want the value D[N][M]

		a	t	c	a	t	g
a							
c	2	1	1	1	2	3	4
a	3	2	2	2	1	2	3
t							
a							
g							

If we only want the value D[N][M]

		a	t	c	a	t	g
a			╲	╲		╲	╲
c		╲	╲		╲	╲	╲
a	3	2	╲ 2	╲ 2	1	╲ 2	╲ 3
t	4	╲ 3	2	╲ 3	╲ 2	1	╲ 2
a			╲	╲		╲	╲
g		╲	╲	╲	╲	╲	

If we only want the value D[N][M]

		a	t	c	a	t	g
a							
c							
a							
t	4	3	2	3	2	1	2
a	5	4	3	3	3	2	2
g							

If we only want the value D[N][M]

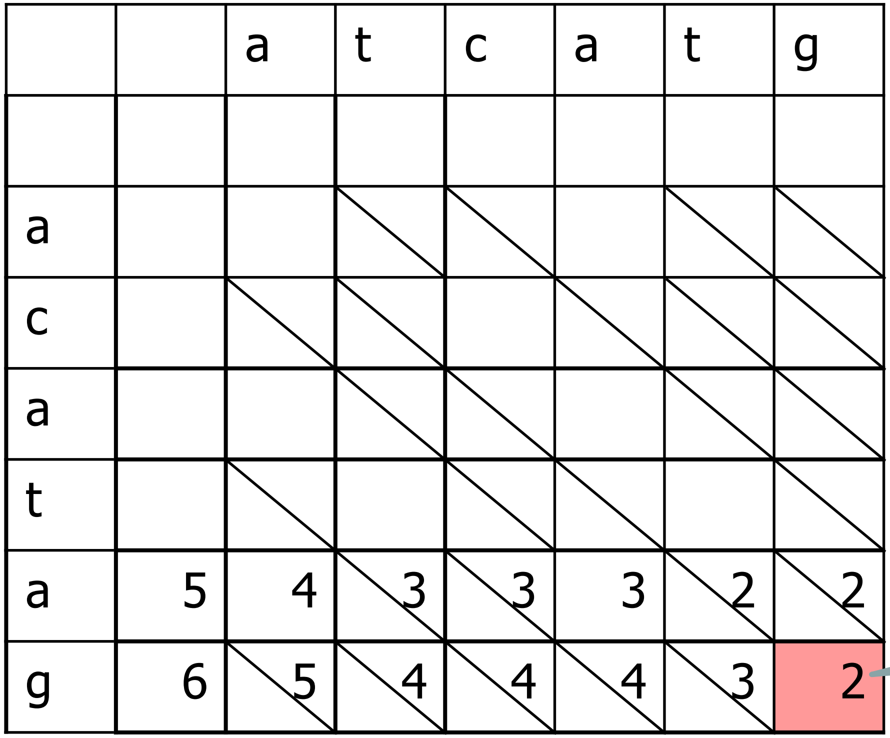
		a	t	c	a	t	g
a			╲	╲		╲	╲
c		╲	╲		╲	╲	╲
a			╲	╲		╲	╲
t		╲		╲	╲		╲
a	5	4	╲ 3	╲ 3	3	╲ 2	╲ 2
g	6	╲ 5	╲ 4	╲ 4	╲ 4	╲ 3	2

Then we don't need more space than to store 2 rows of a table:

because for computing row *i* we only need to know values in row *i*-1, so the values in rows before *i*-1 can be discarded

This computation can be performed in linear space O(*N*)

But in order to actually find a series of edit operations

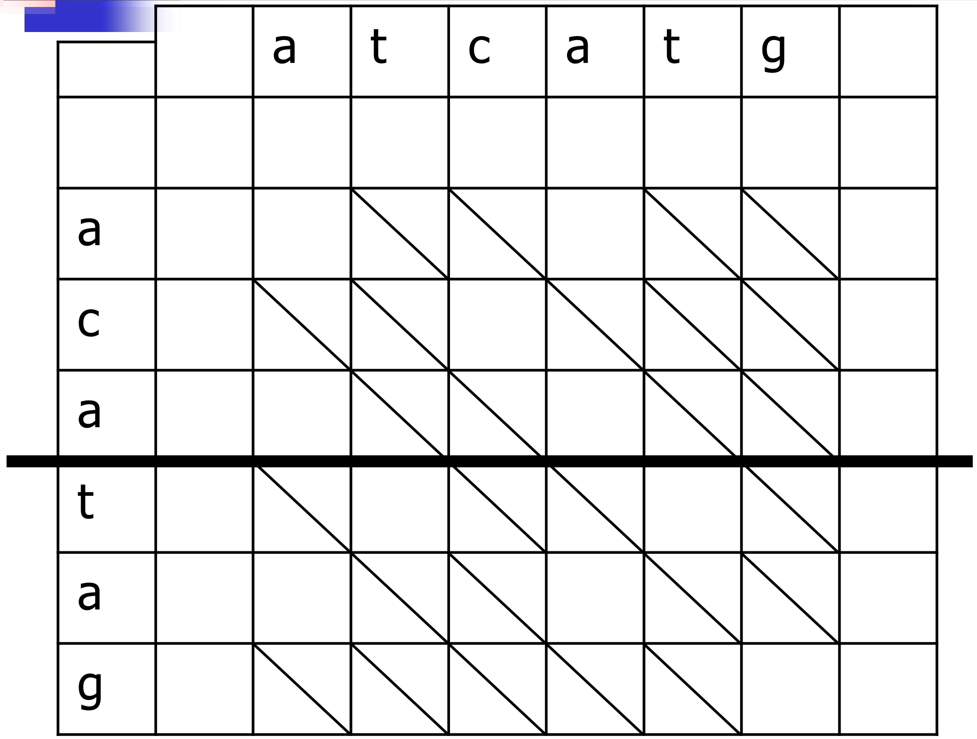


		a	t	c	a	t	g
a			╲	╲		╲	╲
c		╲	╲		╲	╲	╲
a			╲	╲		╲	╲
t		╲		╲	╲		╲
a	5	4	╲ 3	╲ 3	3	╲ 2	╲ 2
g	6	╲ 5	╲ 4	╲ 4	╲ 4	╲ 3	🟥 2

We need to store the pointers for each vertex in the entire graph in order to be able to trace the path back

How did we get here with D=2?

Idea: median border

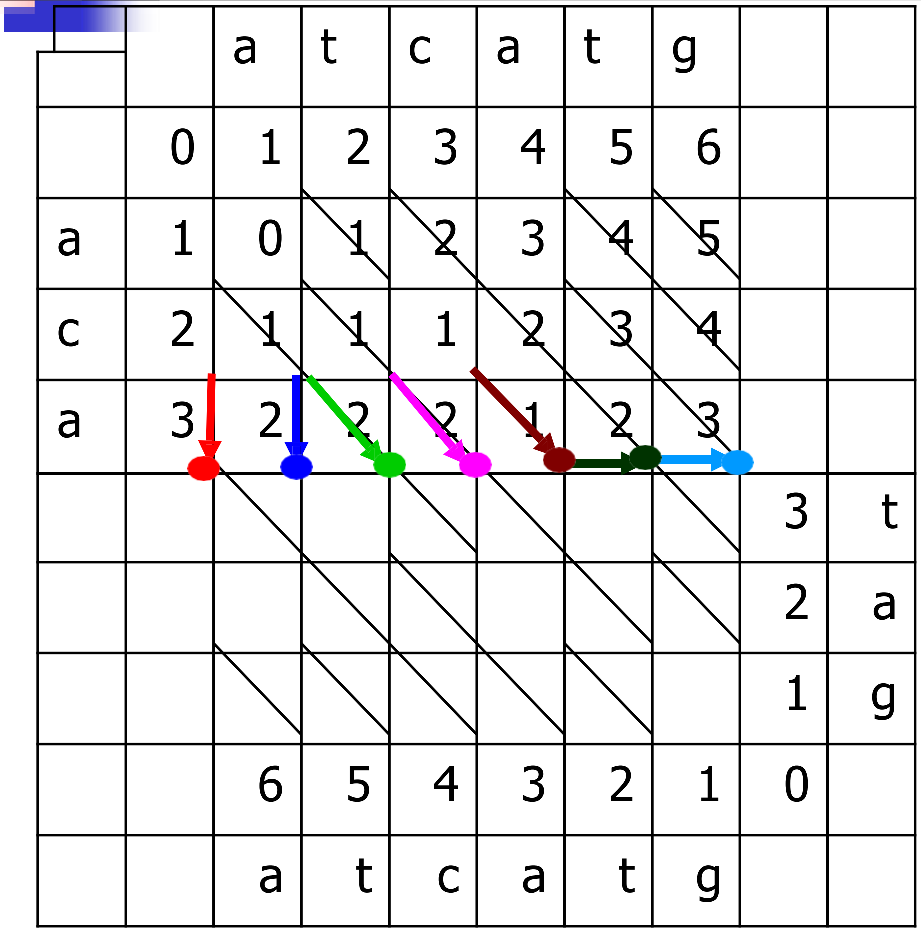


Let us set the median line after the row N/2

Each path, including the optimal path we are looking for, crosses the median line

The goal – to find the point in the median line, where an optimal path crosses it

All the paths from vertex (0,0)…



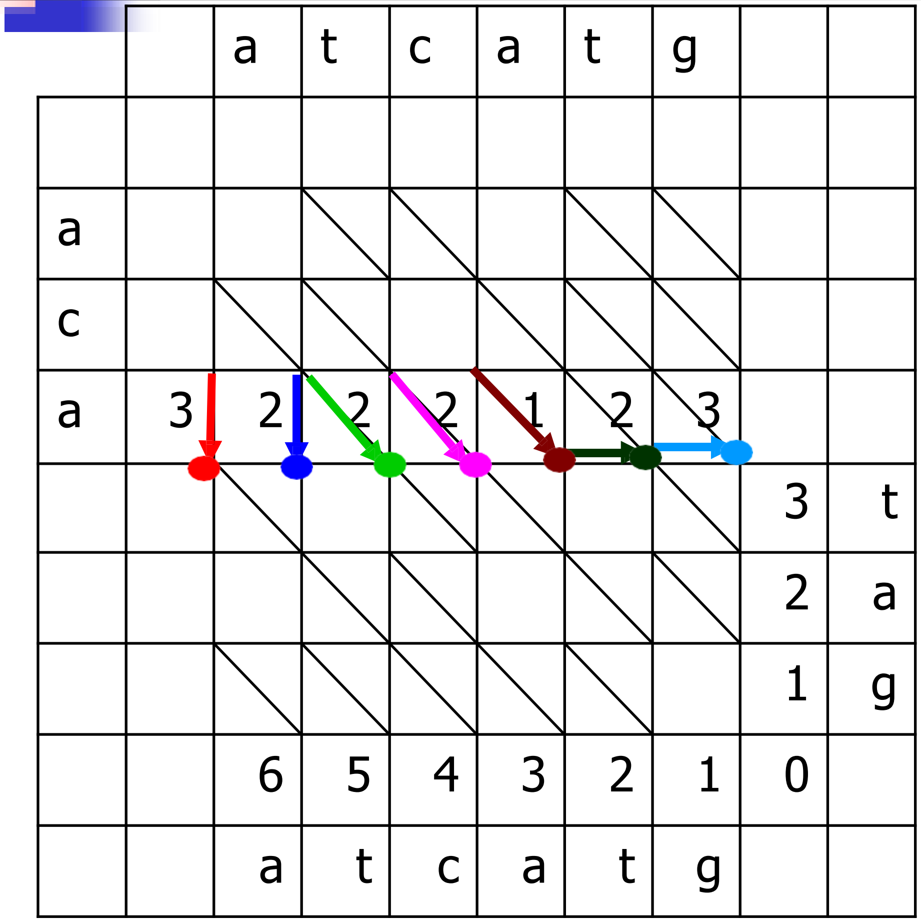
		a	t	c	a	t	g		
	0	1	2	3	4	5	6		
a	1	0	1	2	3	4	5		
c	2	1	1	1	2	3	4		
a	3	2	2	2	1	2	3		
								3	t
								2	a
								1	g
		6	5	4	3	2	1	0	
		a	t	c	a	t	g		

Compute the values of D for the row above the median line

Do not store all the pointers, use only space for 2 rows

Mark the last row with the traceback pointers to the previous row

All the paths from vertex (0,0)…

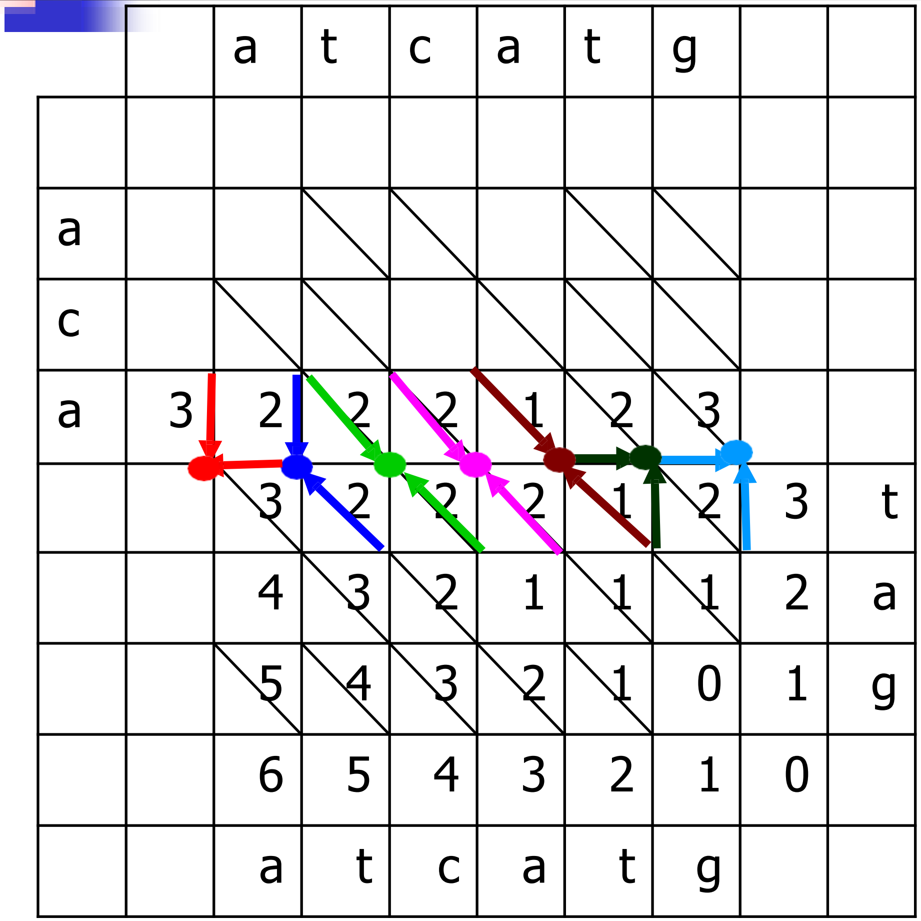


		a	t	c	a	t	g		
a									
c									
a	3	2	2	2	1	2	3		
								3	t
								2	a
								1	g
		6	5	4	3	2	1	0	
		a	t	c	a	t	g		

We have obtained the set of values of the best paths which run from the source till each point on the median line

But we do not know yet which of these points belong to an overall cheapest path

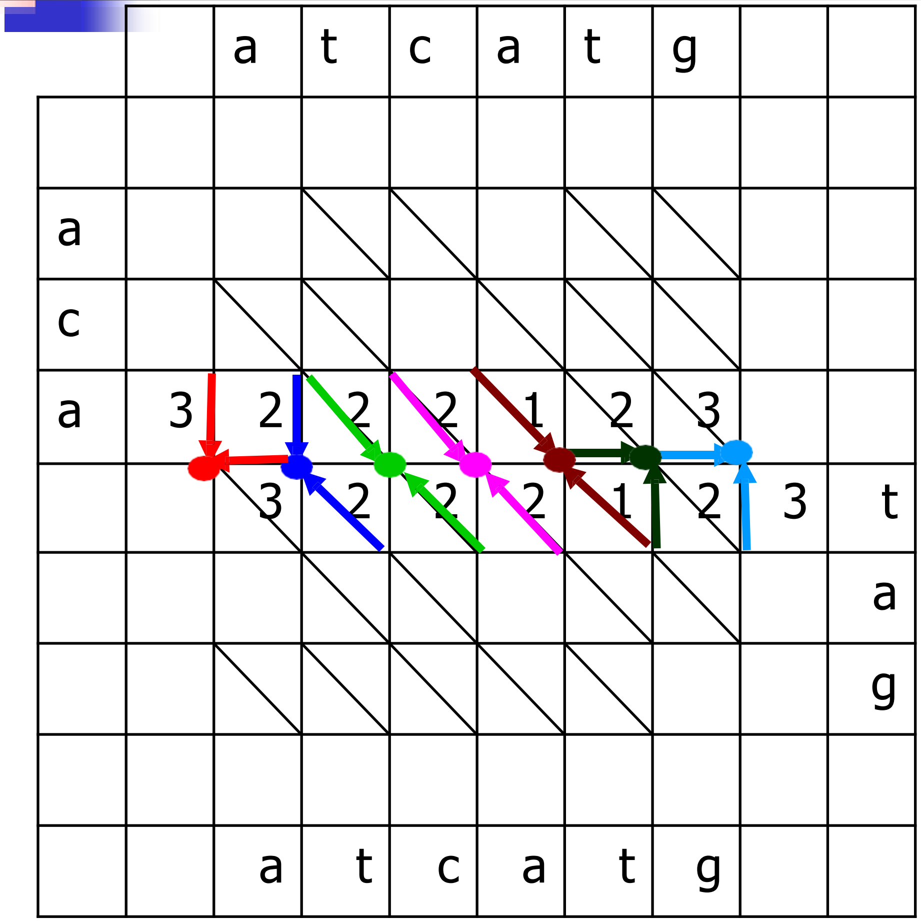
All the paths backwards from vertex (n,m)



		a	t	c	a	t	g		
a									
c									
a	3	2	2	2	1	2	3		
		3	2	2	2	1	2	3	t
		4	3	2	1	1	1	2	a
		5	4	3	2	1	0	1	g
		6	5	4	3	2	1	0	
		a	t	c	a	t	g		

Next, we compute all the best paths running from the destination point (6,6) till each point on the median line, considering the same strings in the opposite direction

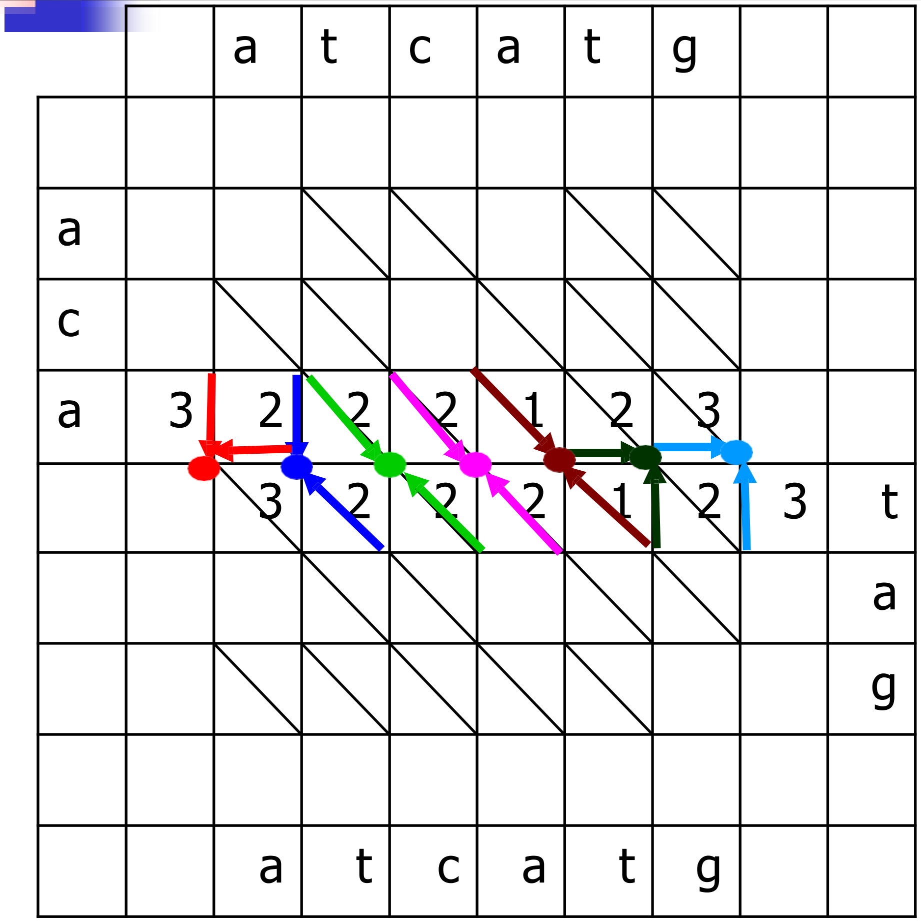
All the paths running from (0,0) and from (n,m)



Now we have enough information to compute the total cost of the cheapest path passing through each point on the median line:

- ● 3+3=6
- ● 2+2=4
- ● 2+2=4
- ● 2+2=4
- ● 1+1=2
- ● 2+2=4
- ● 3+3=6

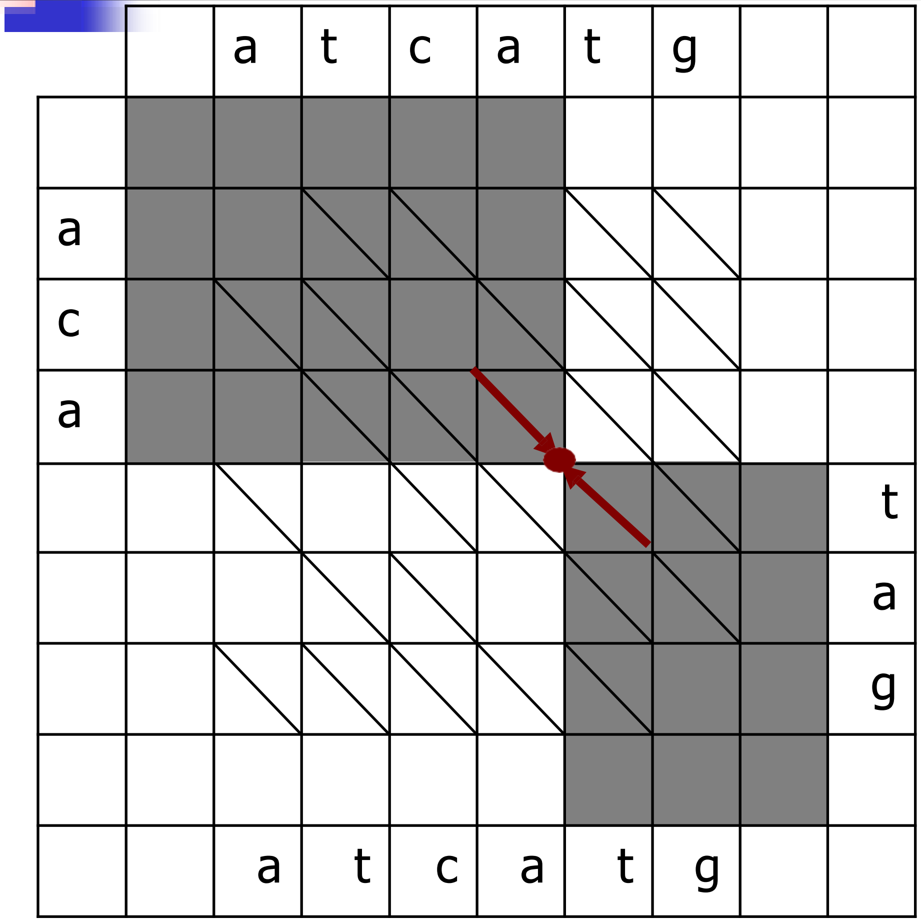
The point where best path hits the median line



We infer that the overall cheapest path of cost 2 hits the median line at vertex (3,4):

- ● 3+3=6
- ● 2+2=4
- ● 2+2=4
- ● 2+2=4
- ● 1+1=2
- ● 2+2=4
- ● 3+3=6

The remaining parts of the path

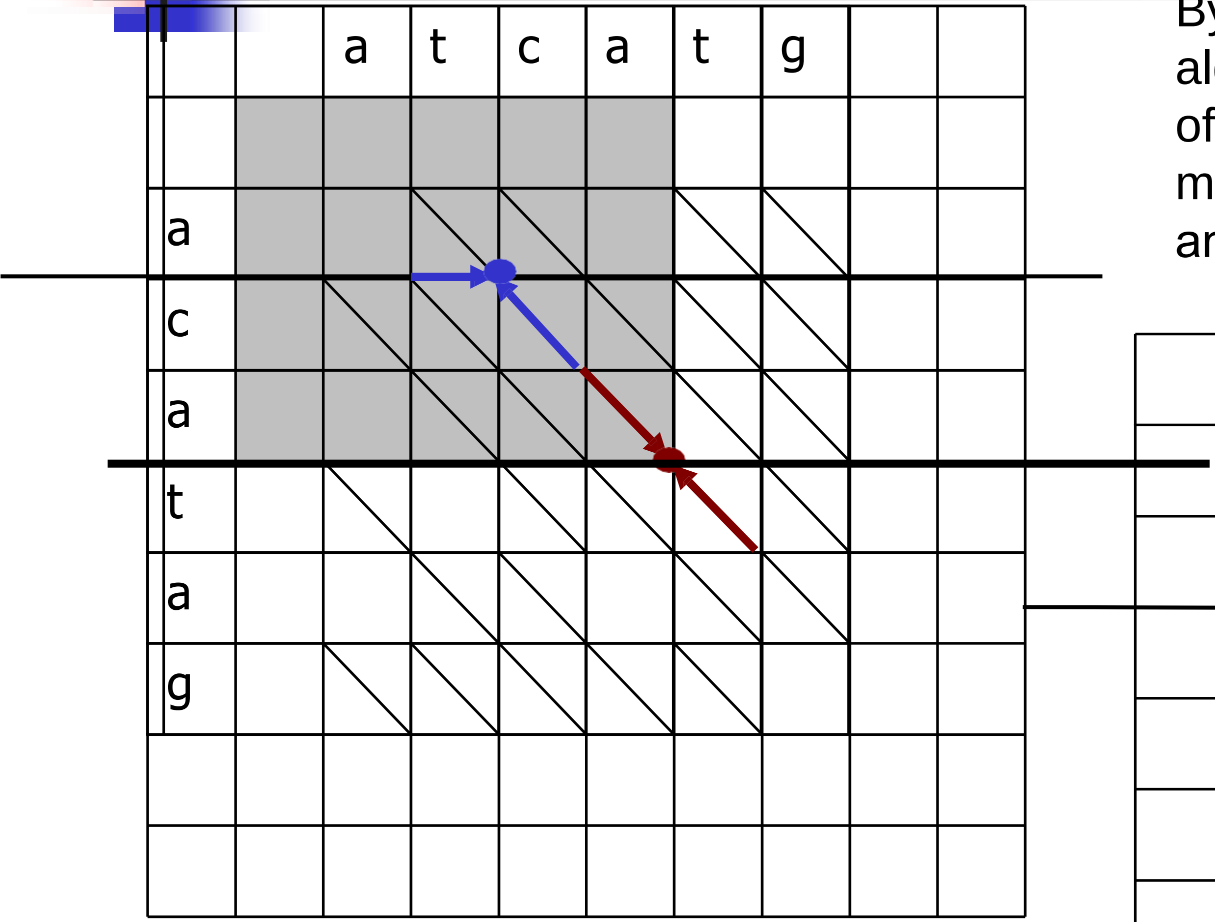


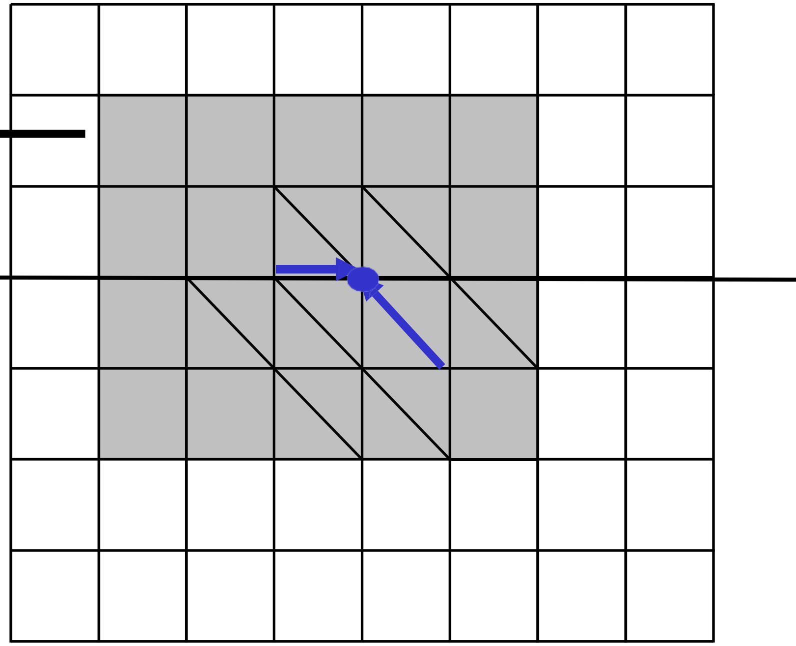
Thus, we found one vertex on the best path

Next, we need to find the remaining parts of this path which can pass only inside grey areas of the grid

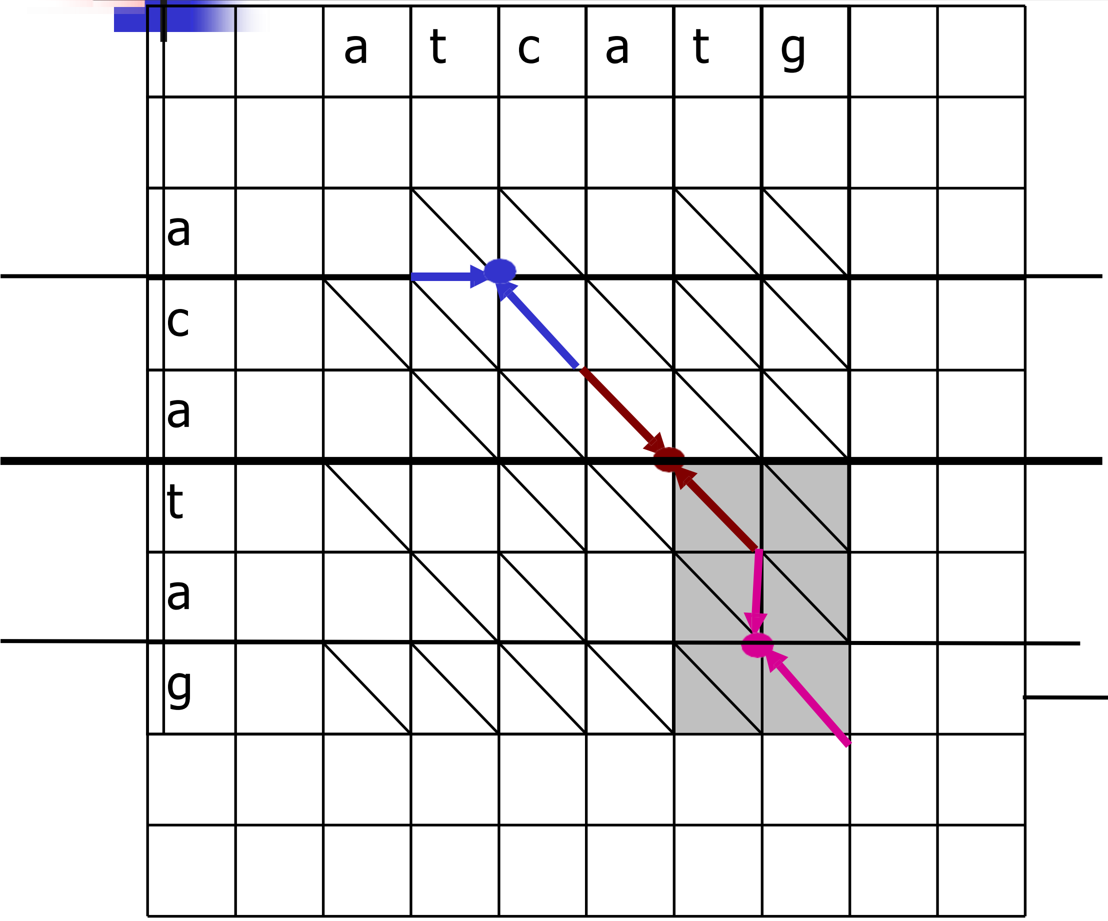
Recursive computation for NM/2 cells

By the same bi-directional algorithm we compute the piece of the path which hits the median line of the upper-left and …

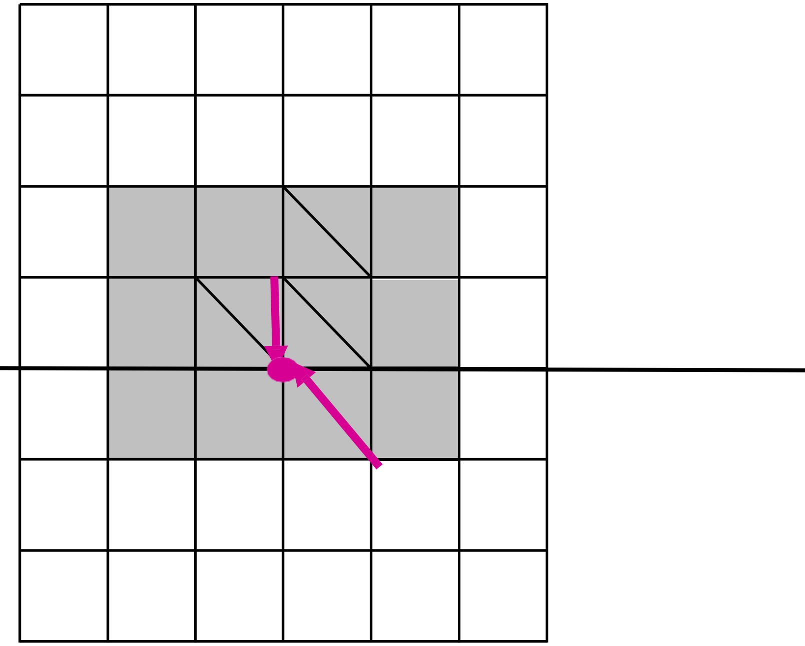




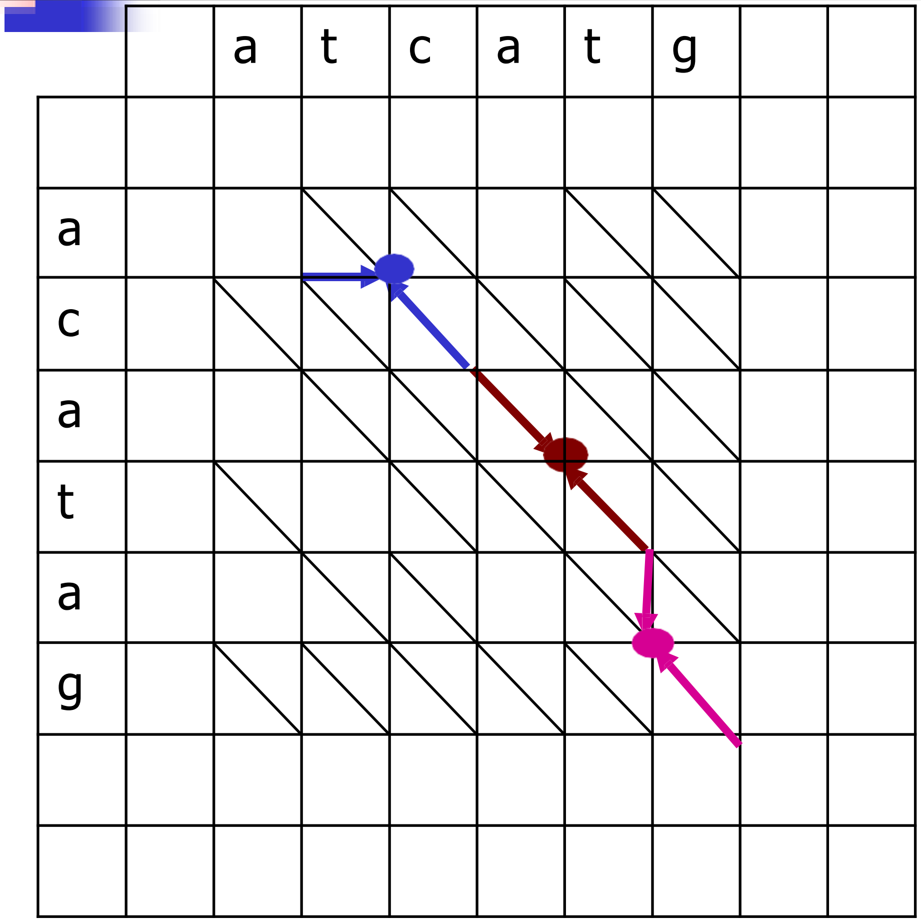
Recursive computation for NM/2 cells



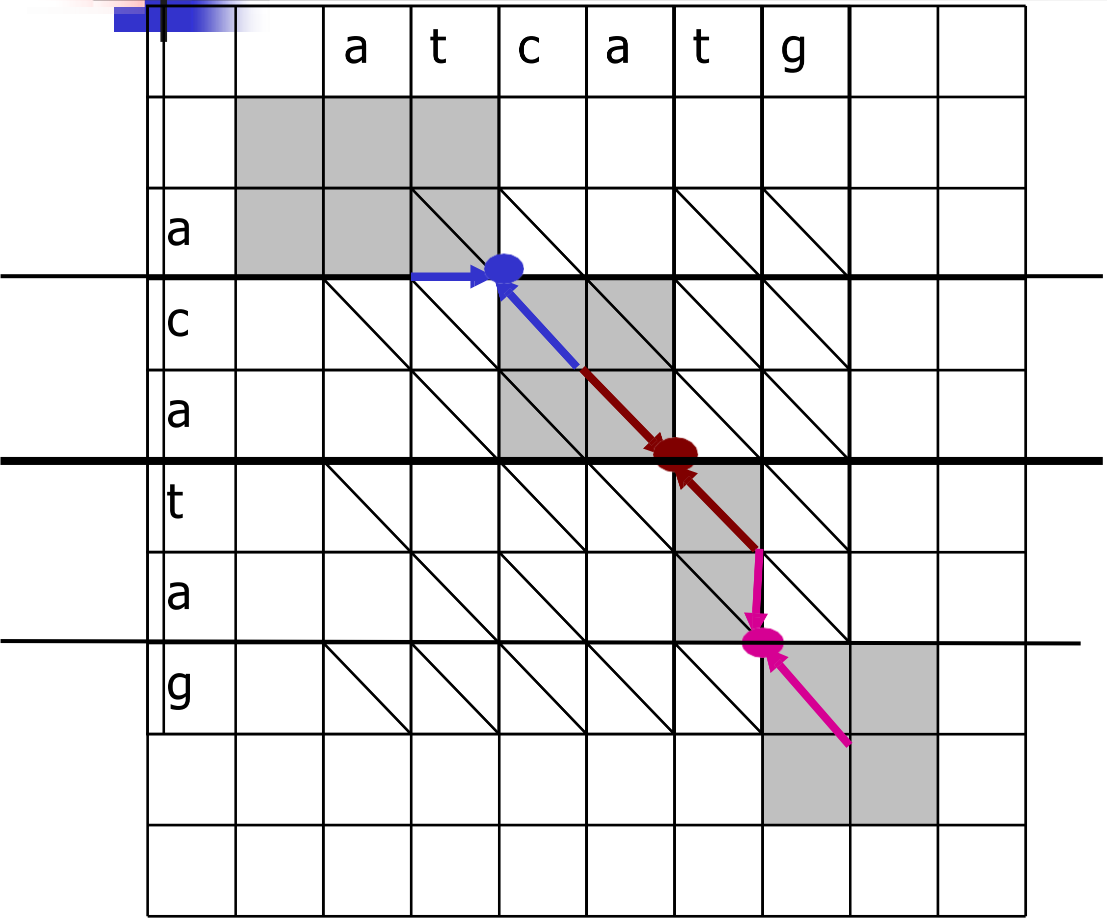
By the same bi-directional algorithm we compute the piece of the path which hits the median line of the upper-left and … the bottom-right parts



We have 1+2 pieces of the best path

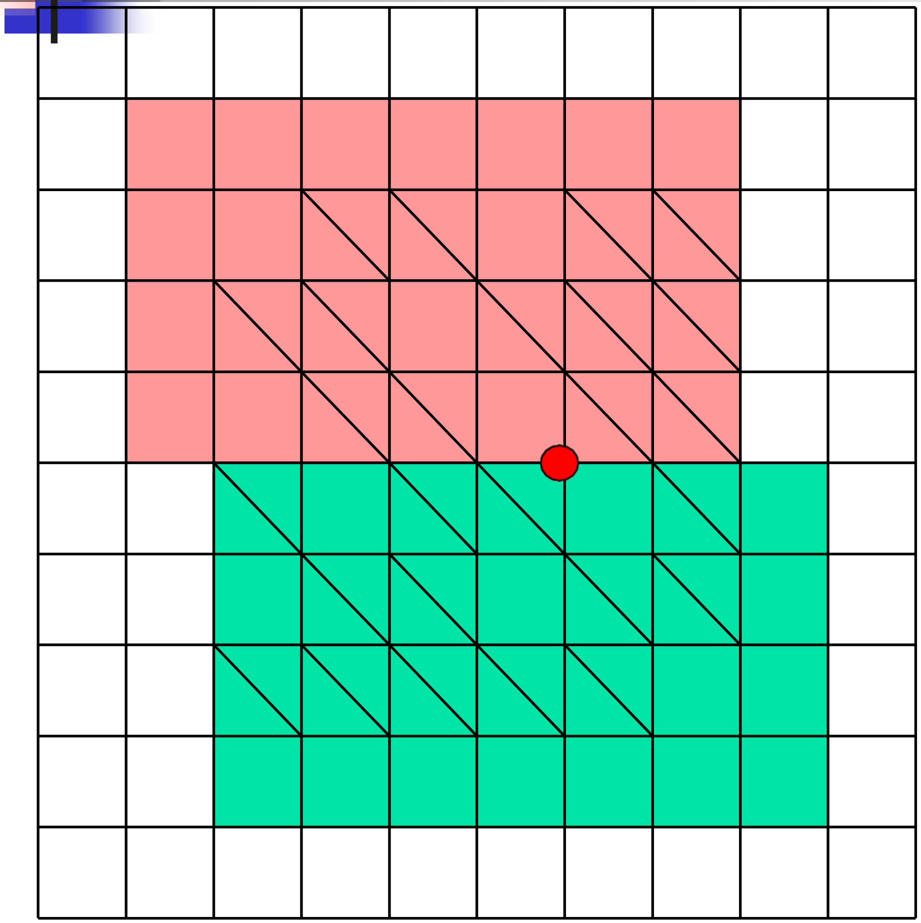


Recursive computation for NM/4



The remaining areas are in grey

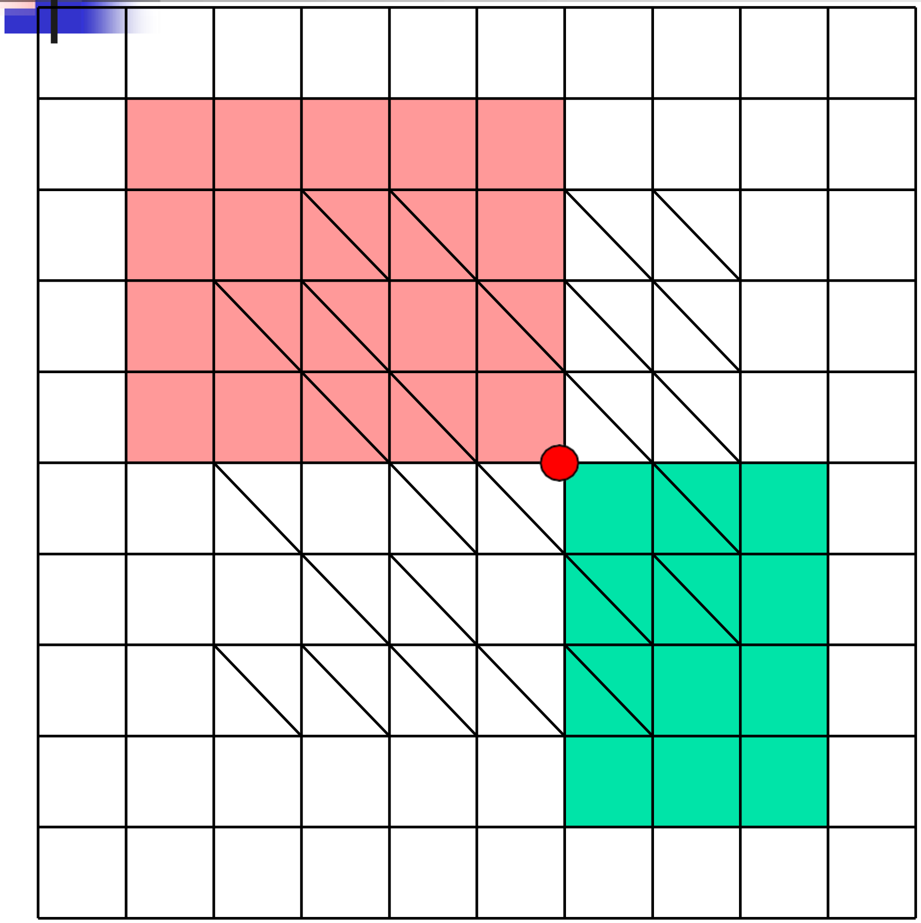
The Hirschberg's algorithm. Computed NM cells



Each time we compute two tables, whose total size is 2 times smaller than in the previous step.

In each recursive computation we find an additional point belonging to the cheapest path, and record it

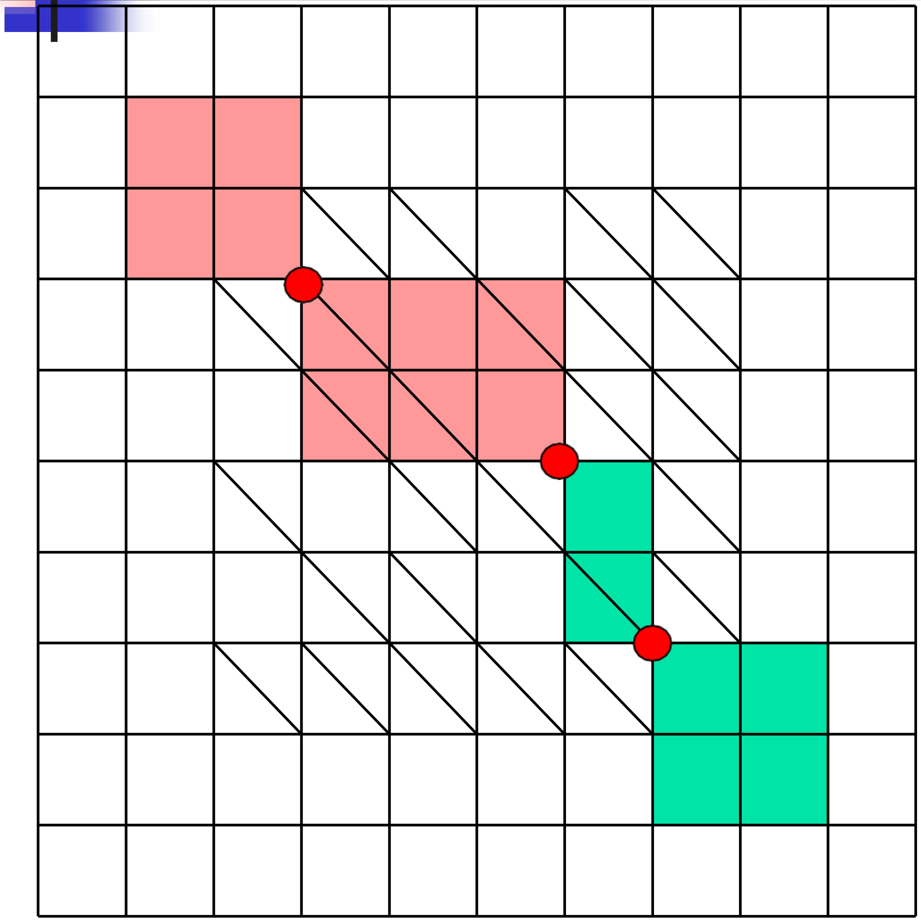
The Hirschberg's algorithm. Computed NM/2 cells



Each time we compute two tables, whose total size is 2 times smaller than in the previous step.

In each recursive computation we find an additional point belonging to the cheapest path, and record it

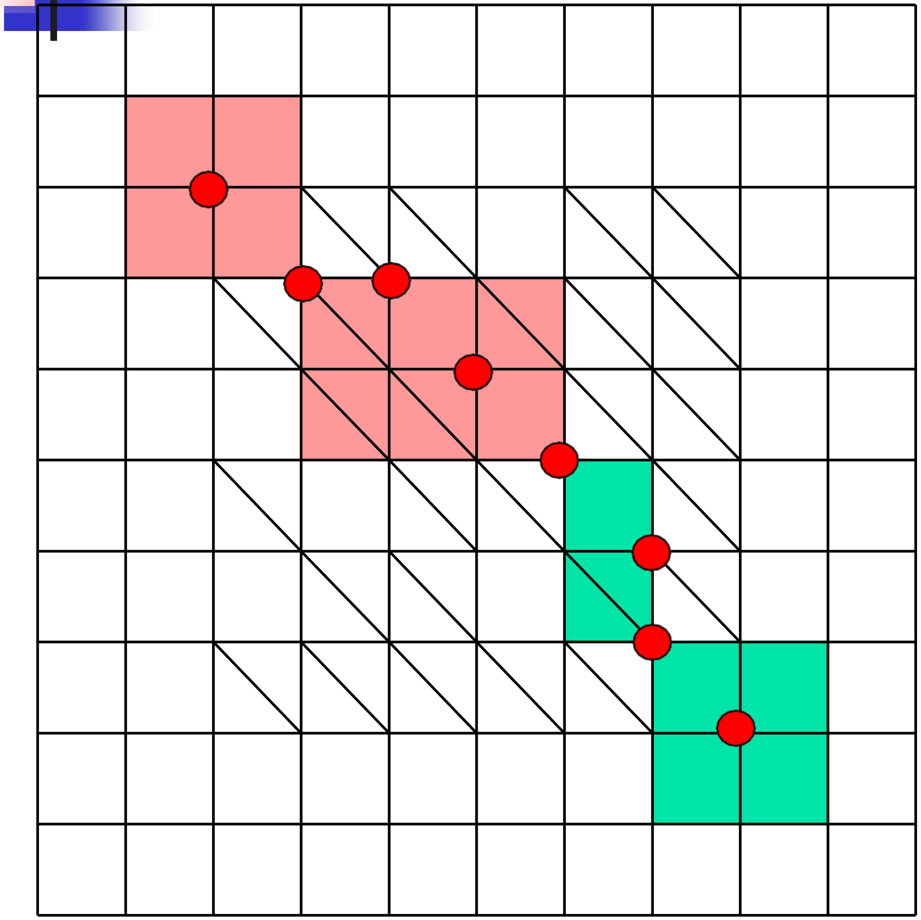
The Hirschberg’s algorithm. Computed NM/4 cells



Each time we compute two tables, whose total size is 2 times smaller than in the previous step.

In each recursive computation we find an additional point belonging to the cheapest path, and record it

The Hirschberg’s algorithm. Termination



When only 2 rows left to be computed, we can find the best path using only 2 rows of each table

The total path is complete

The Hirschberg's algorithm. Time complexity

- The algorithm computes values of $NM+NM/2+NM/4+....NM/(NM/2)=$ $= NM(1+1/2+1/4+...)=2NM$ cells

The time complexity is 2NM - still O(NM)

The Hirschberg’s algorithm. Space complexity

- The algorithm never uses the space more than for 2 rows of the table

The space complexity is O(*N*)

The pseudocode of the Hirschberg’s algorithm can be found here

You only need to understand and will be tested about the main idea of the algorithm: not the precise pseudocode