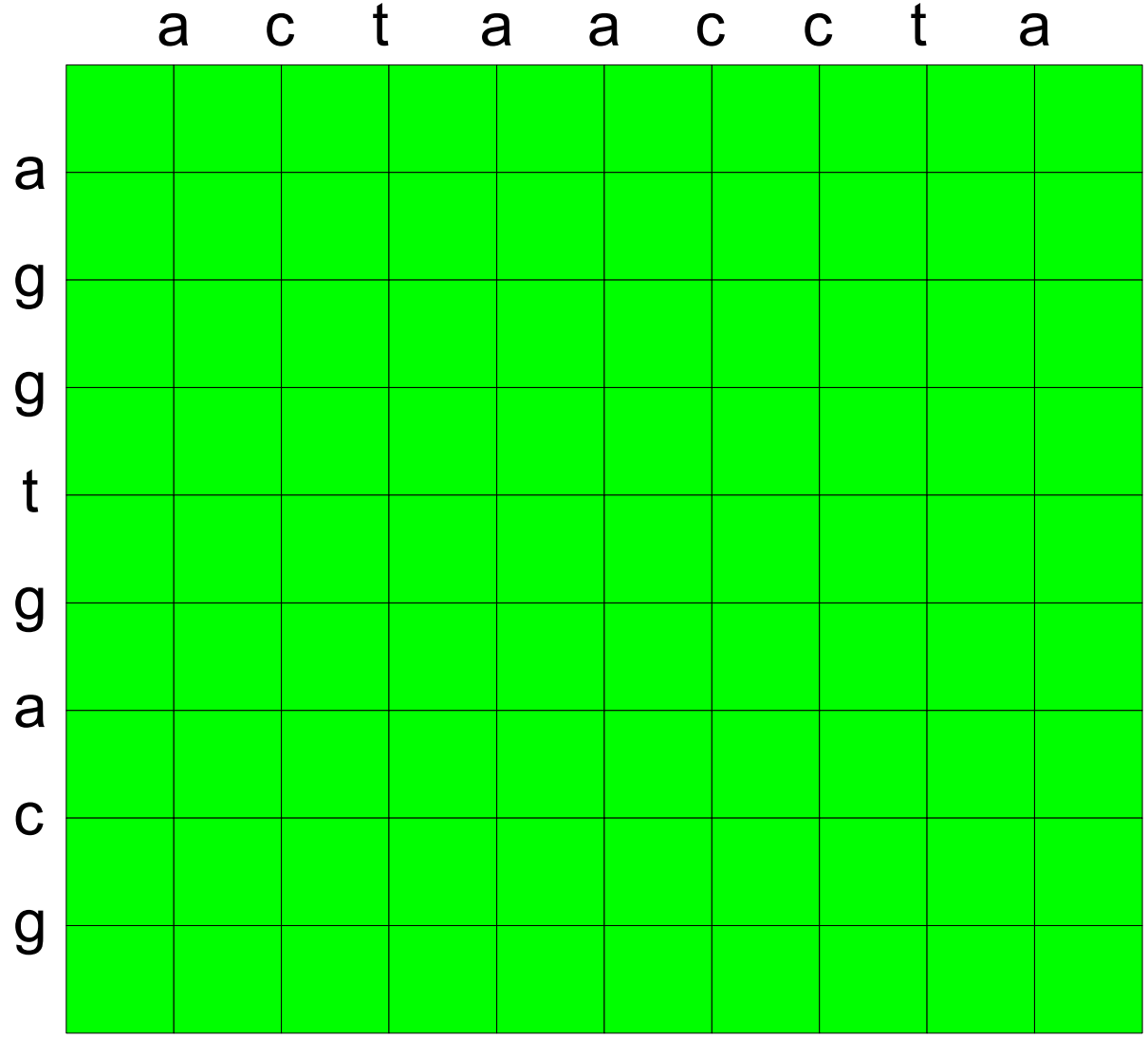
Global alignment in linear space

Global alignment in linear space

Goal: Find an optimal alignment of $A[1..n]$ and $B[1..m]$ in linear space, i.e. $O(n)$

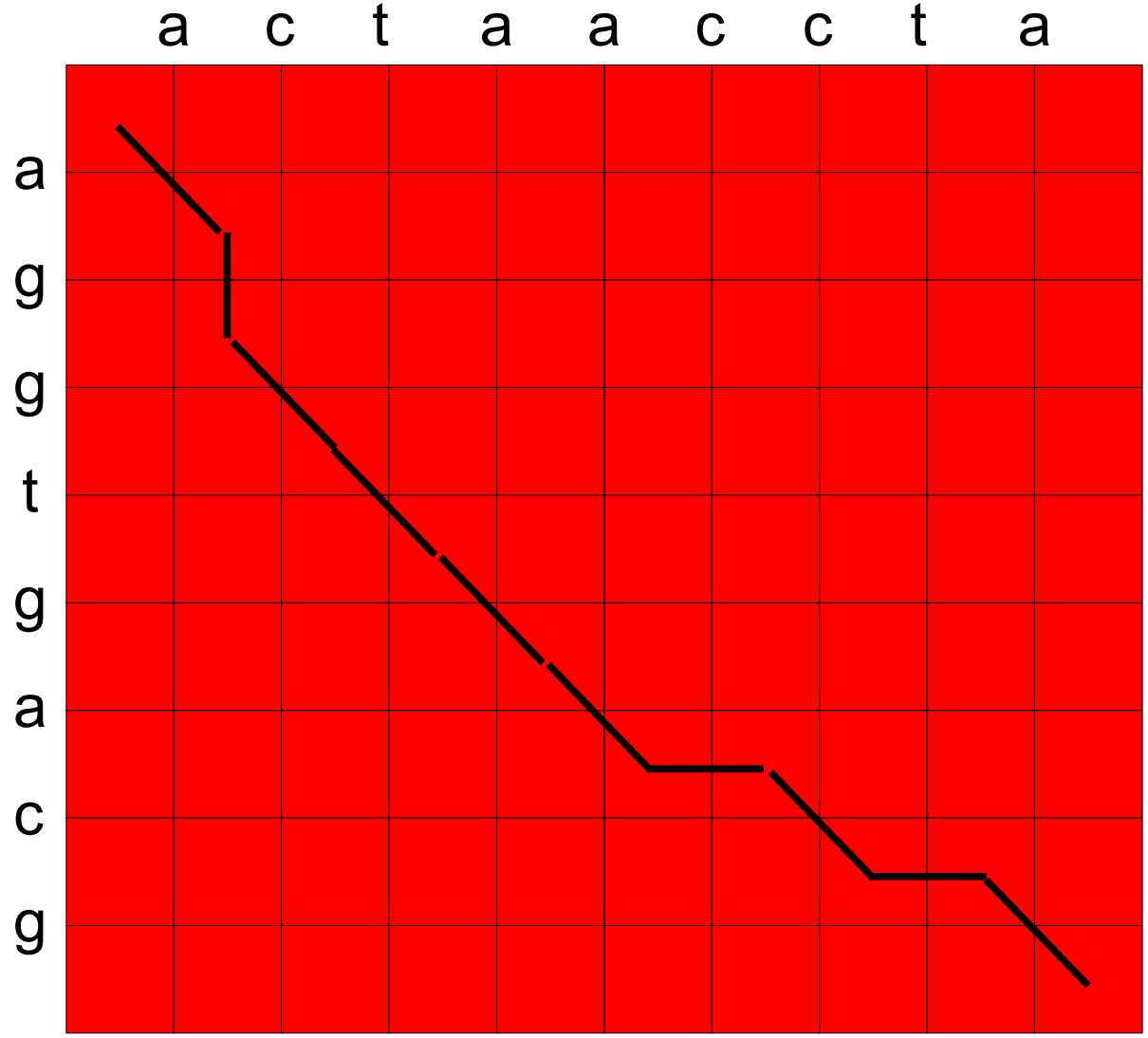


Existing algorithm:

Global alignment with backtracking $O(nm)$ time and space, but the optimal cost can be found in space $O(n)$

Global alignment in linear space

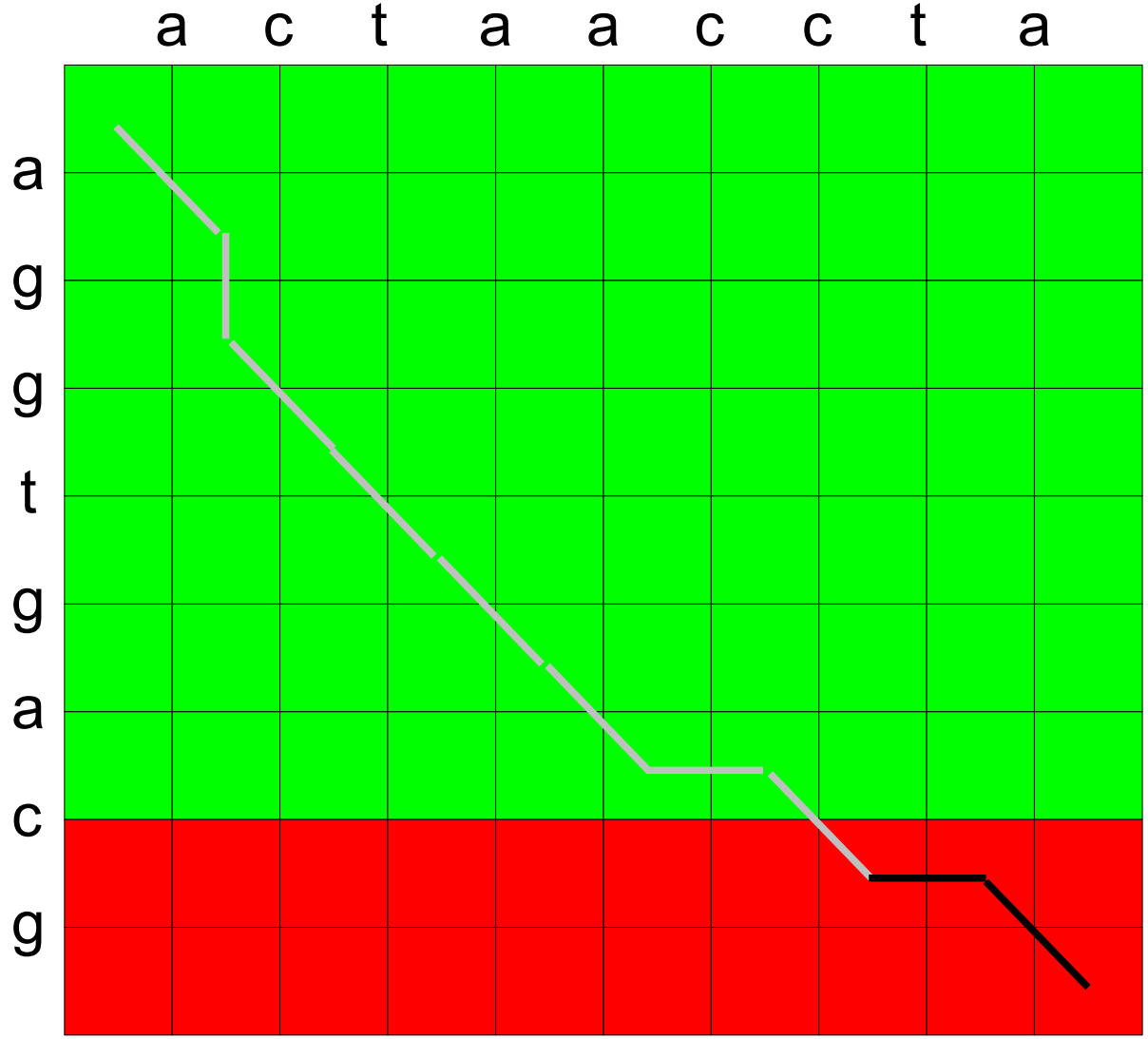
Goal: Find an optimal alignment of $A[1..n]$ and $B[1..m]$ in linear space, i.e. $O(n)$



Existing algorithm:

Global alignment with backtracking $O(nm)$ time and space, but the optimal cost can be found in space $O(n)$

Linear space – First idea



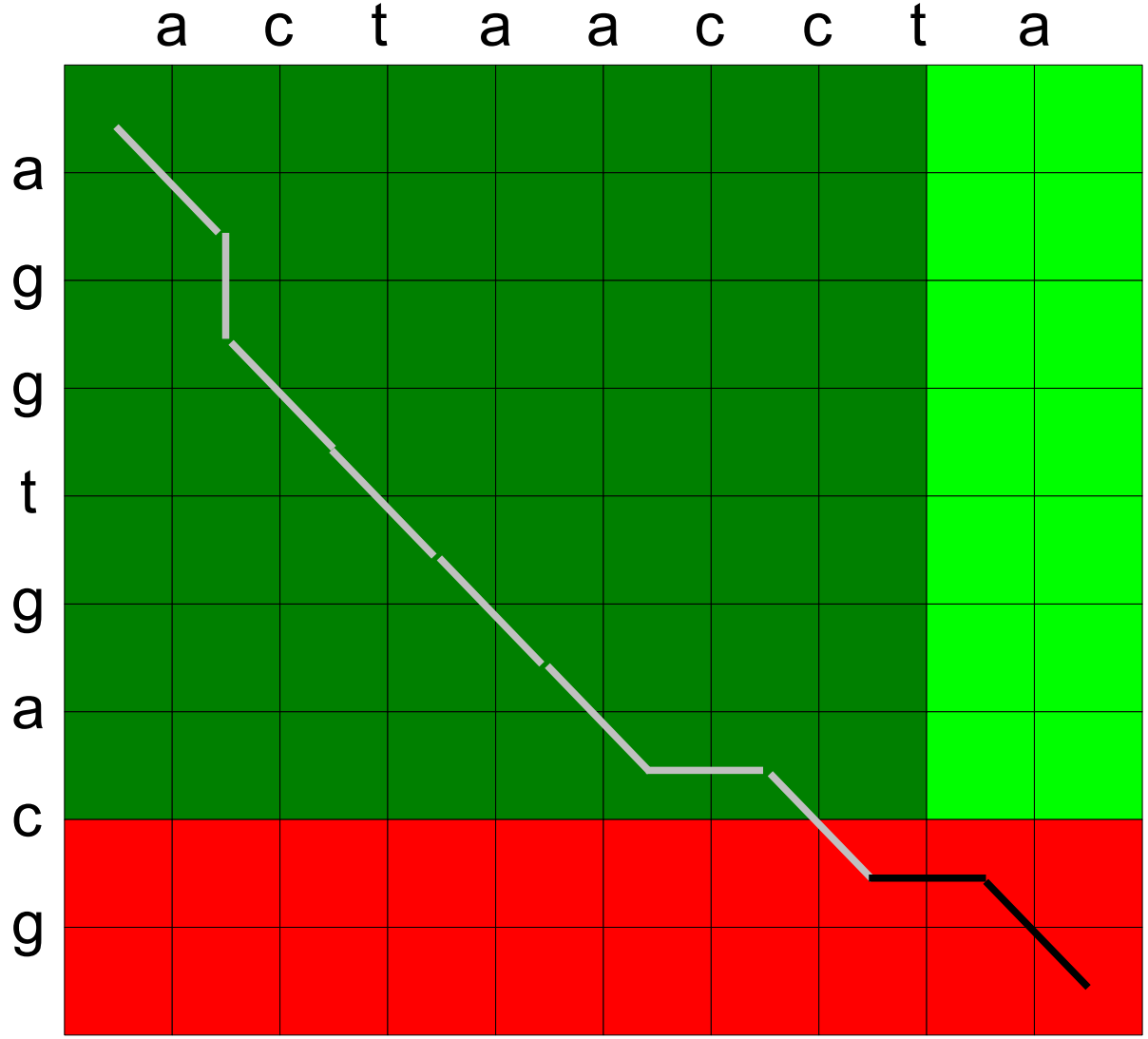
First idea:

Keep only two rows in memory and recompute when other parts of the matrix is needed during backtracking ...

Time:

nm

Linear space – First idea



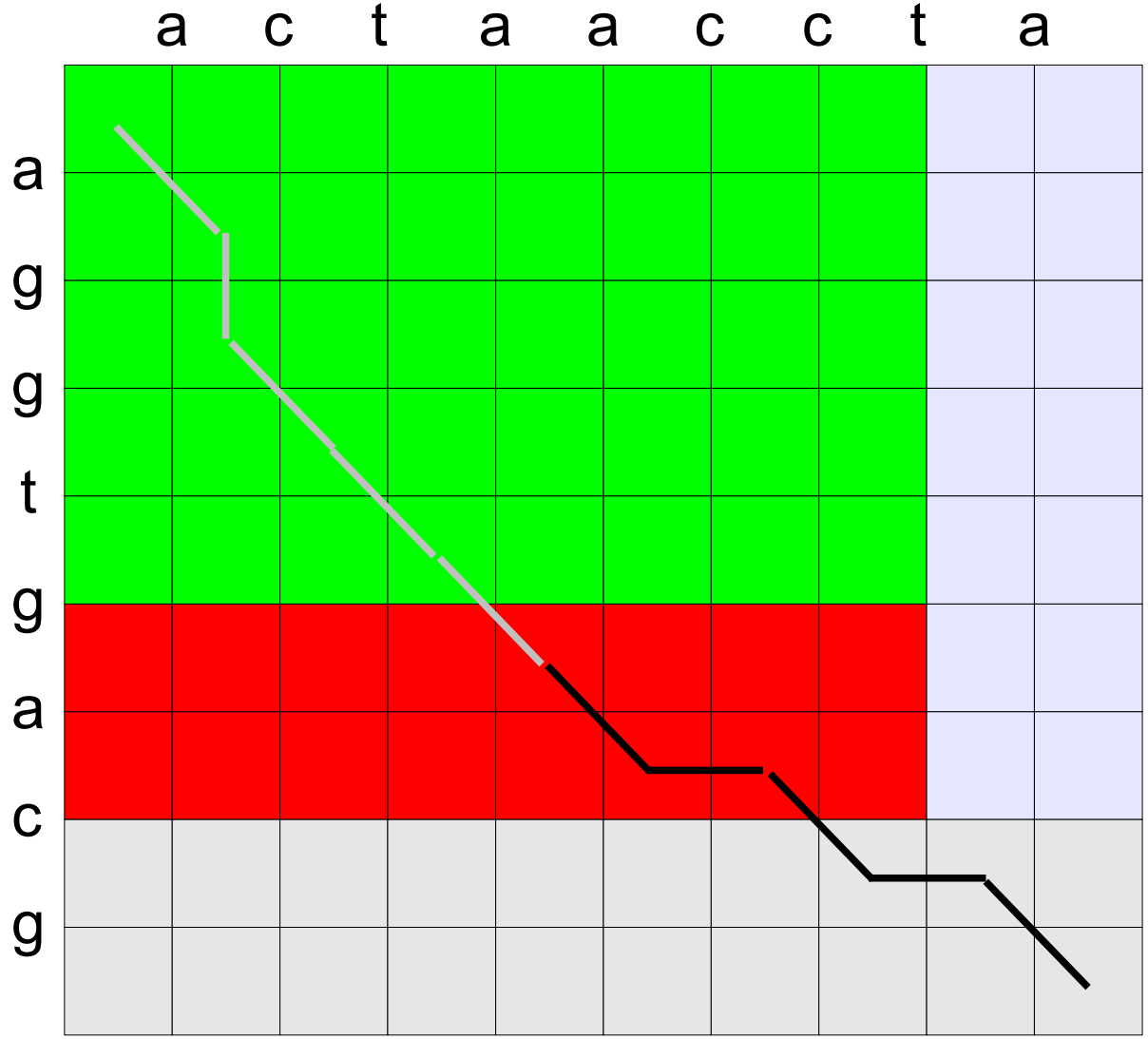
First idea:

Keep only two rows in memory and recompute when other parts of the matrix is needed during backtracking ...

Time:

nm

Linear space – First idea



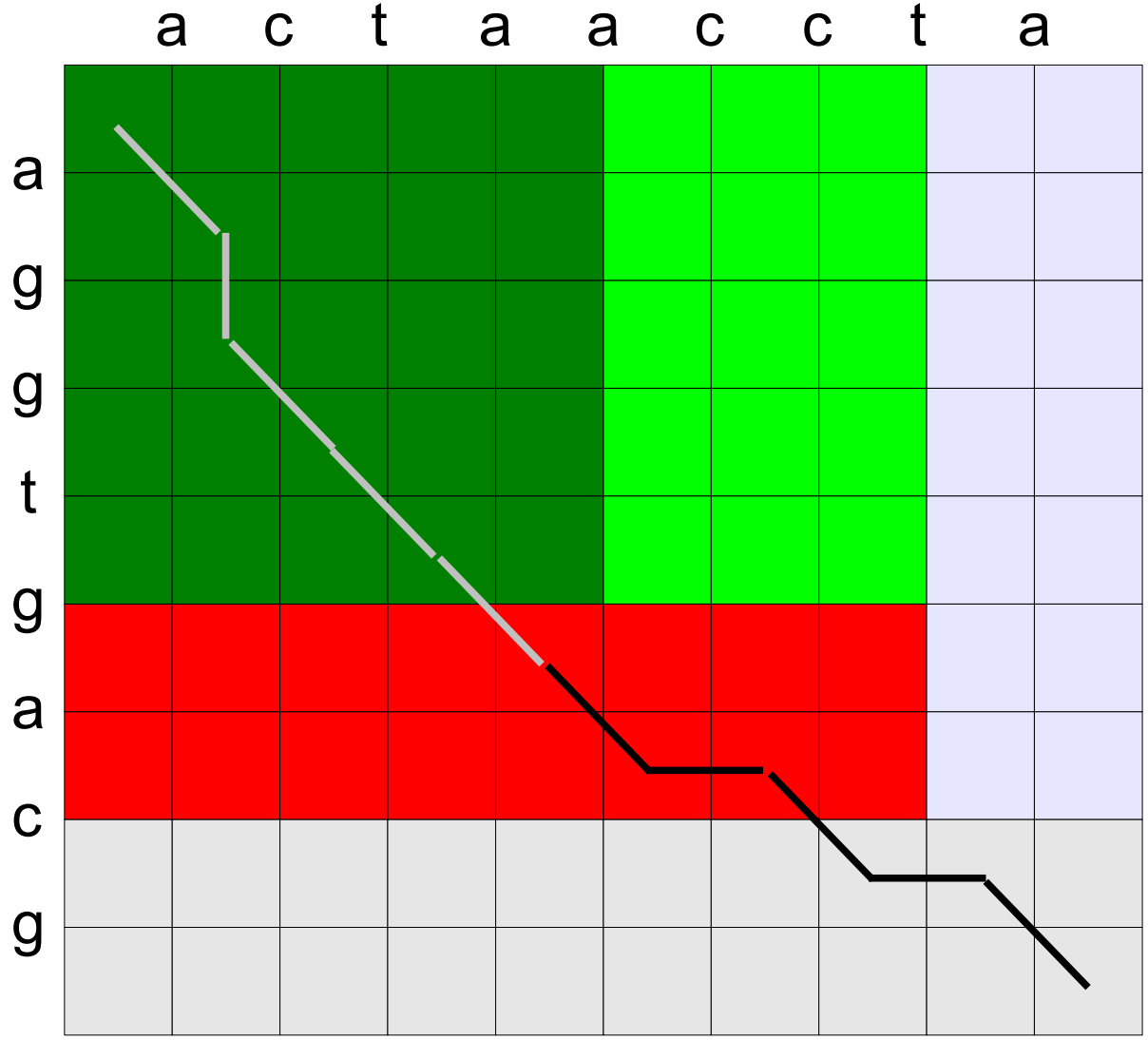
First idea:

Keep only two rows in memory and recompute when other parts of the matrix is needed during backtracking ...

Time:

nm +
(n-2)m +

Linear space – First idea



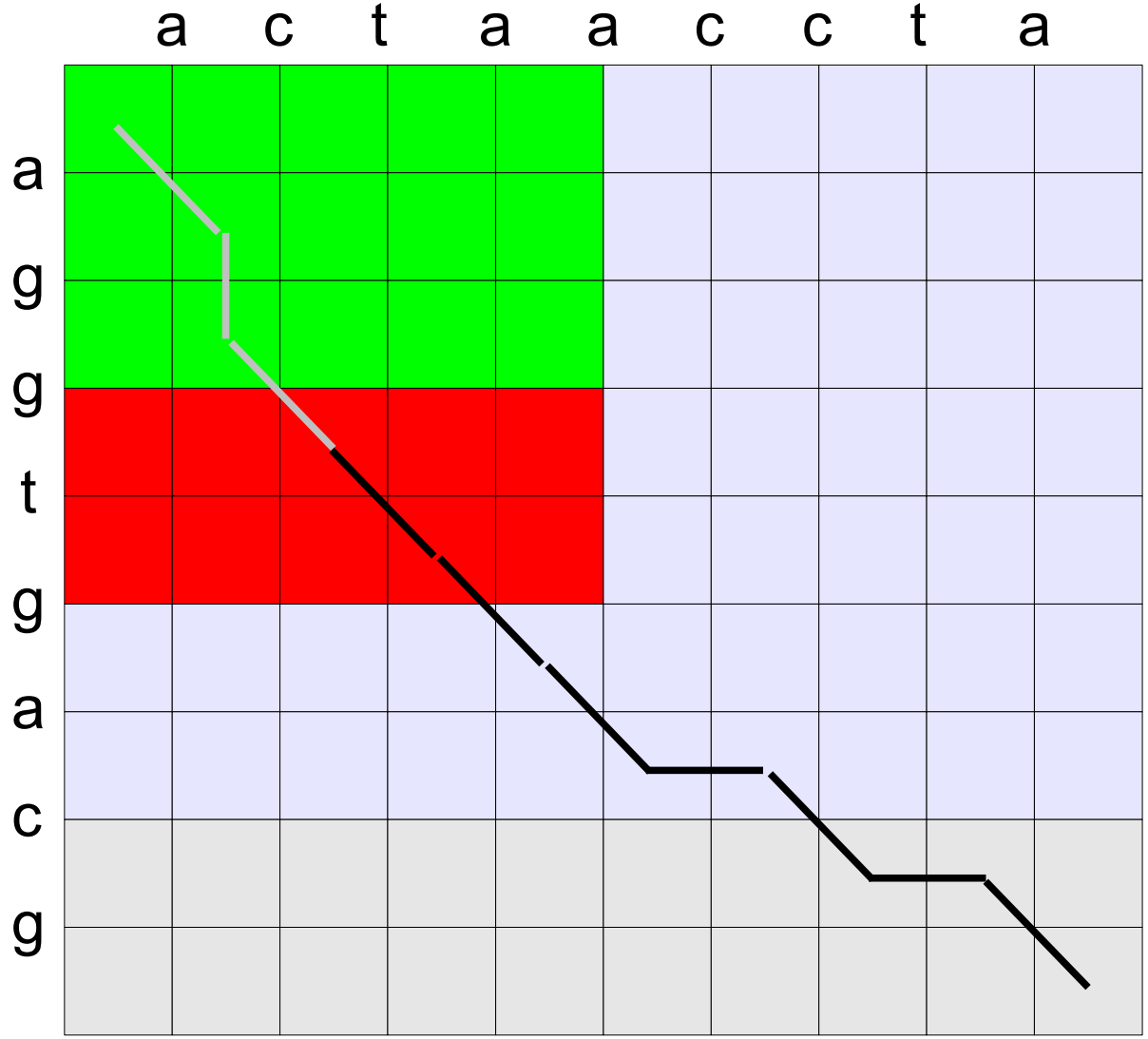
First idea:

Keep only two rows in memory and recompute when other parts of the matrix is needed during backtracking ...

Time:

nm +
(n-2)m +

Linear space – First idea



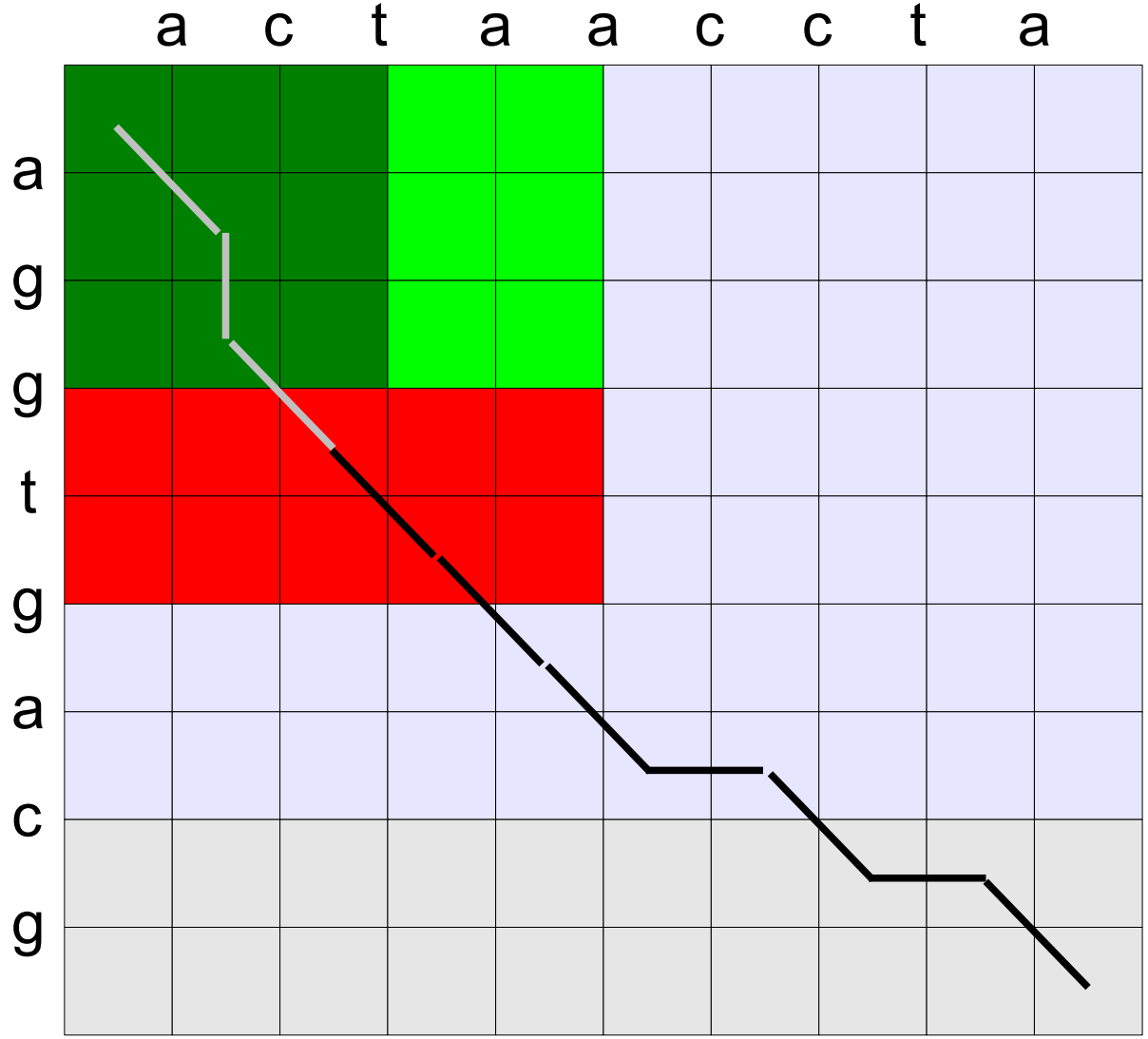
First idea:

Keep only two rows in memory and recompute when other parts of the matrix is needed during backtracking ...

Time:

nm +
(n-2)m +
(n-4)m

Linear space – First idea



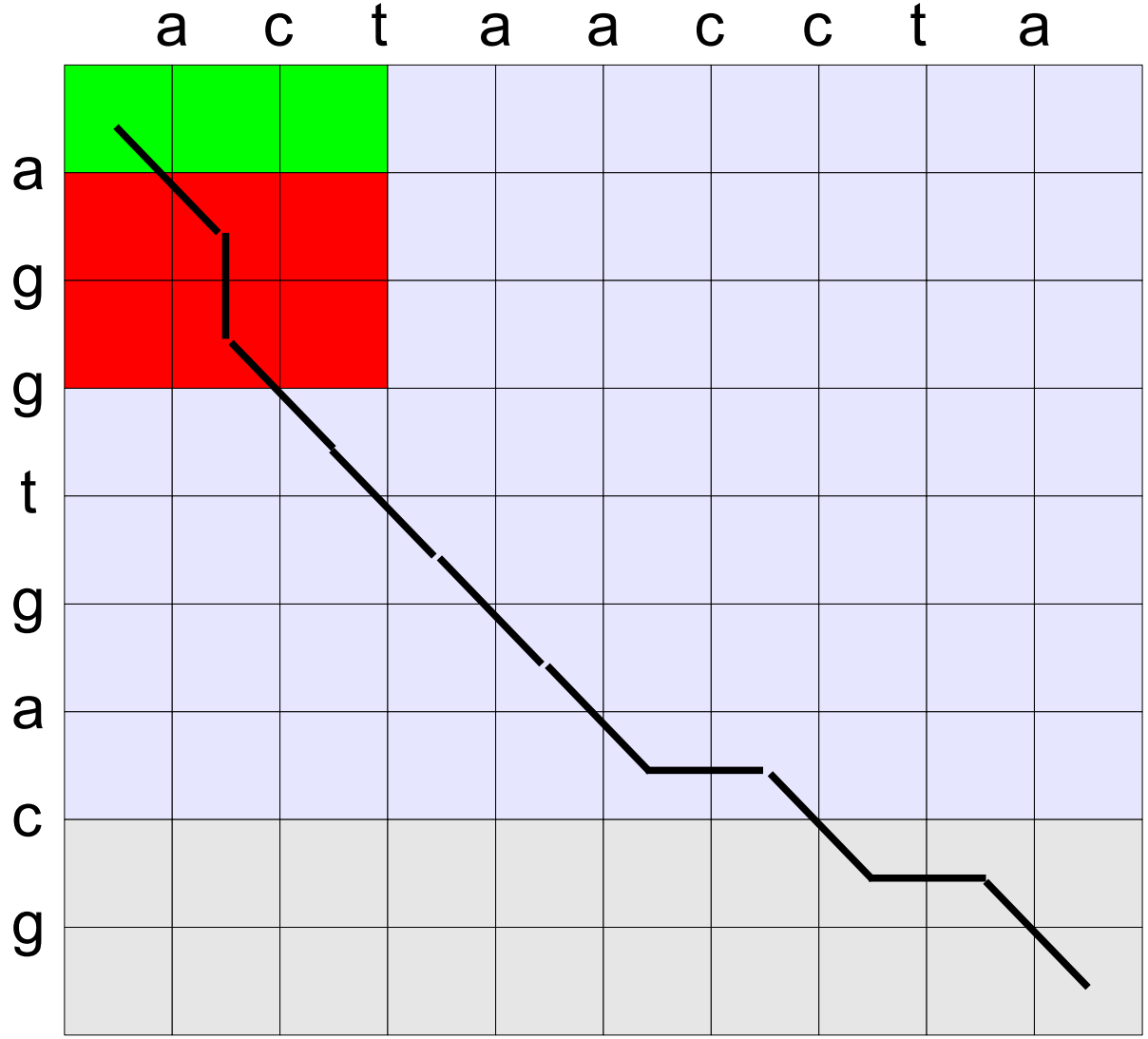
First idea:

Keep only two rows in memory and recompute when other parts of the matrix is needed during backtracking ...

Time:

nm +
(n-2)m +
(n-4)m

Linear space – First idea



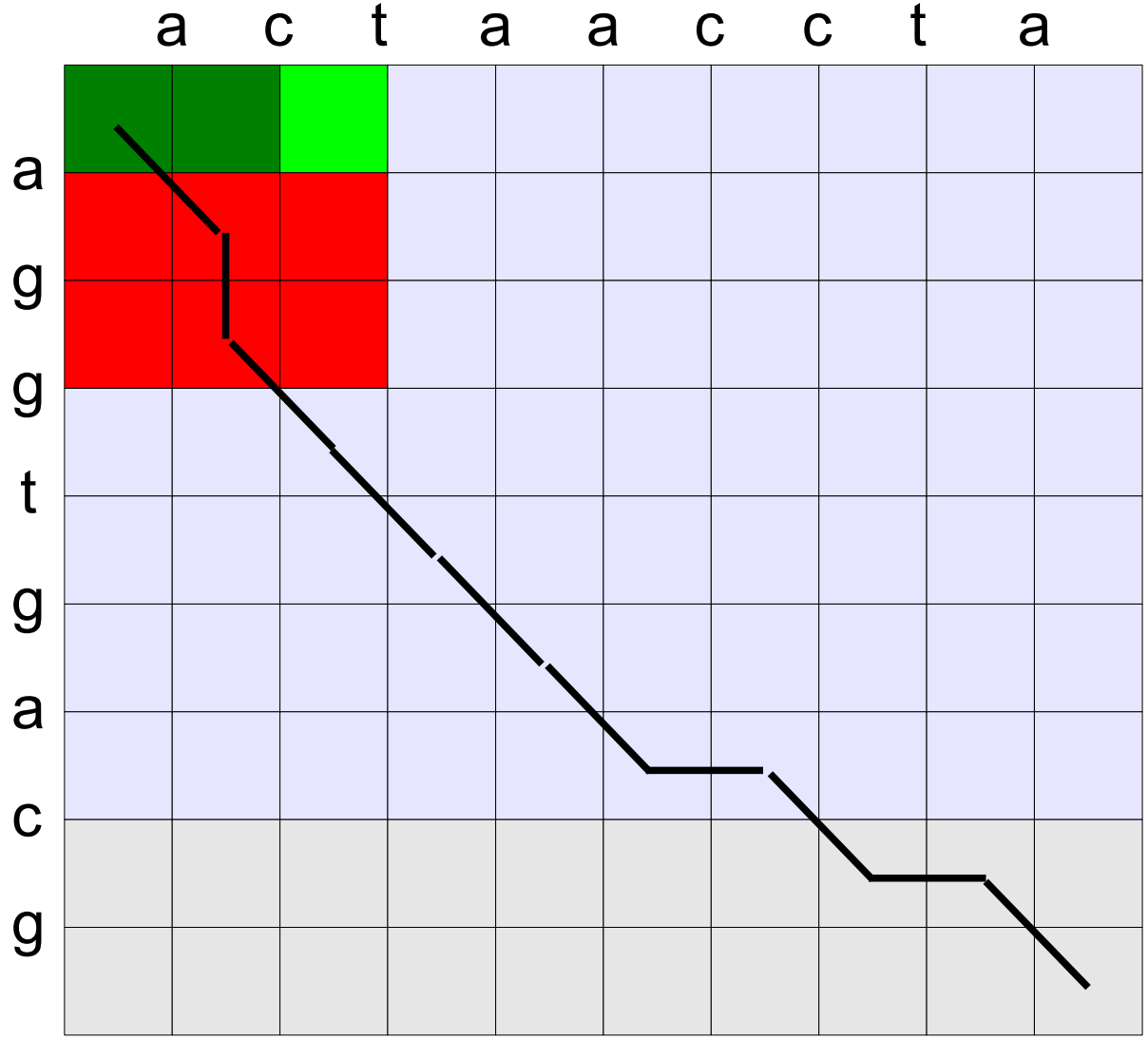
First idea:

Keep only two rows in memory and recompute when other parts of the matrix is needed during backtracking ...

Time:

$nm +$
$(n - 2)m +$
$(n - 4)m +$
$(n - 6)m$

Linear space – First idea



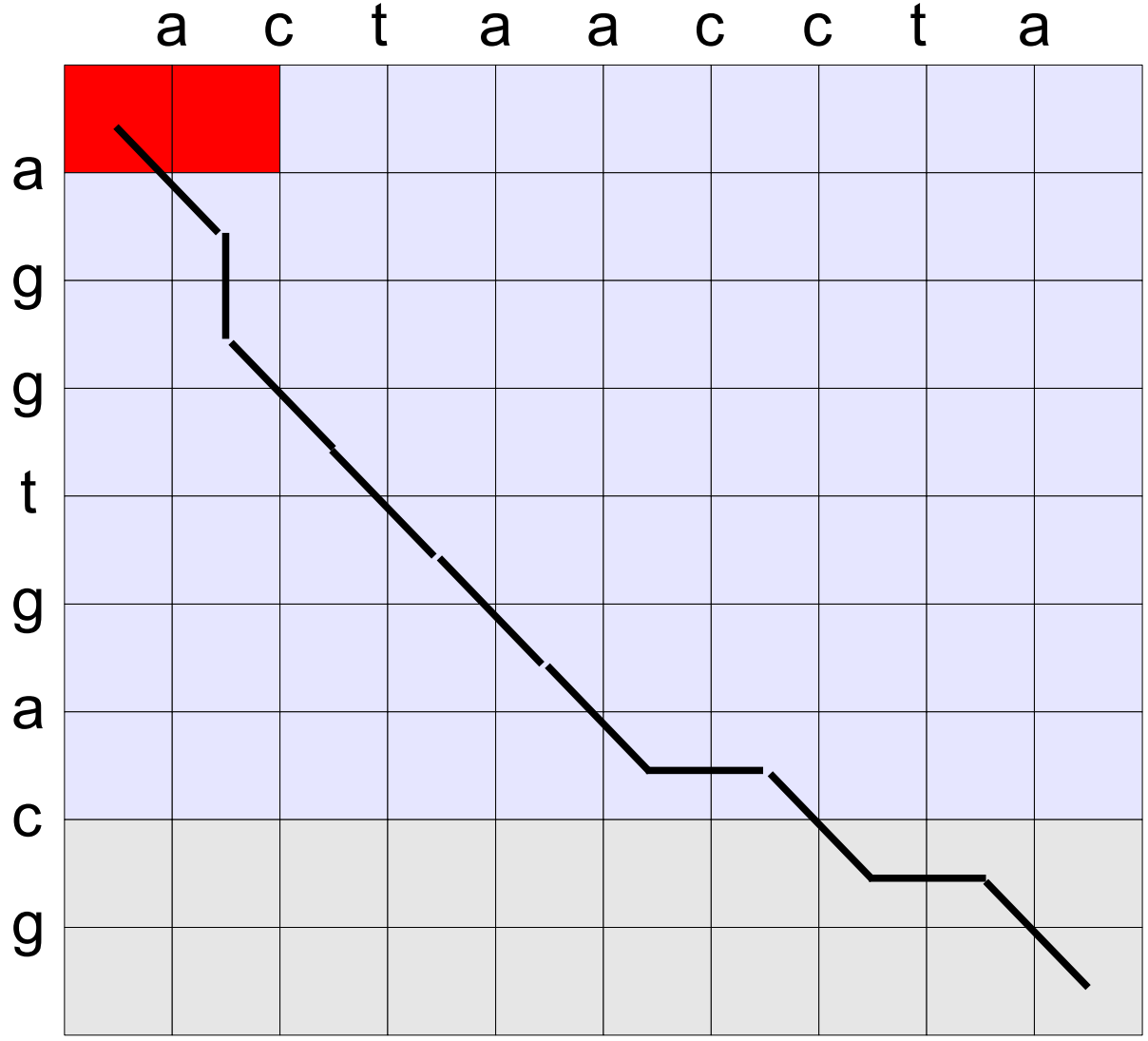
First idea:

Keep only two rows in memory and recompute when other parts of the matrix is needed during backtracking ...

Time:

$nm +$
$(n - 2)m +$
$(n - 4)m +$
$(n - 6)m$

Linear space – First idea



First idea:

Keep only two rows in memory and recompute when other parts of the matrix is needed during backtracking ...

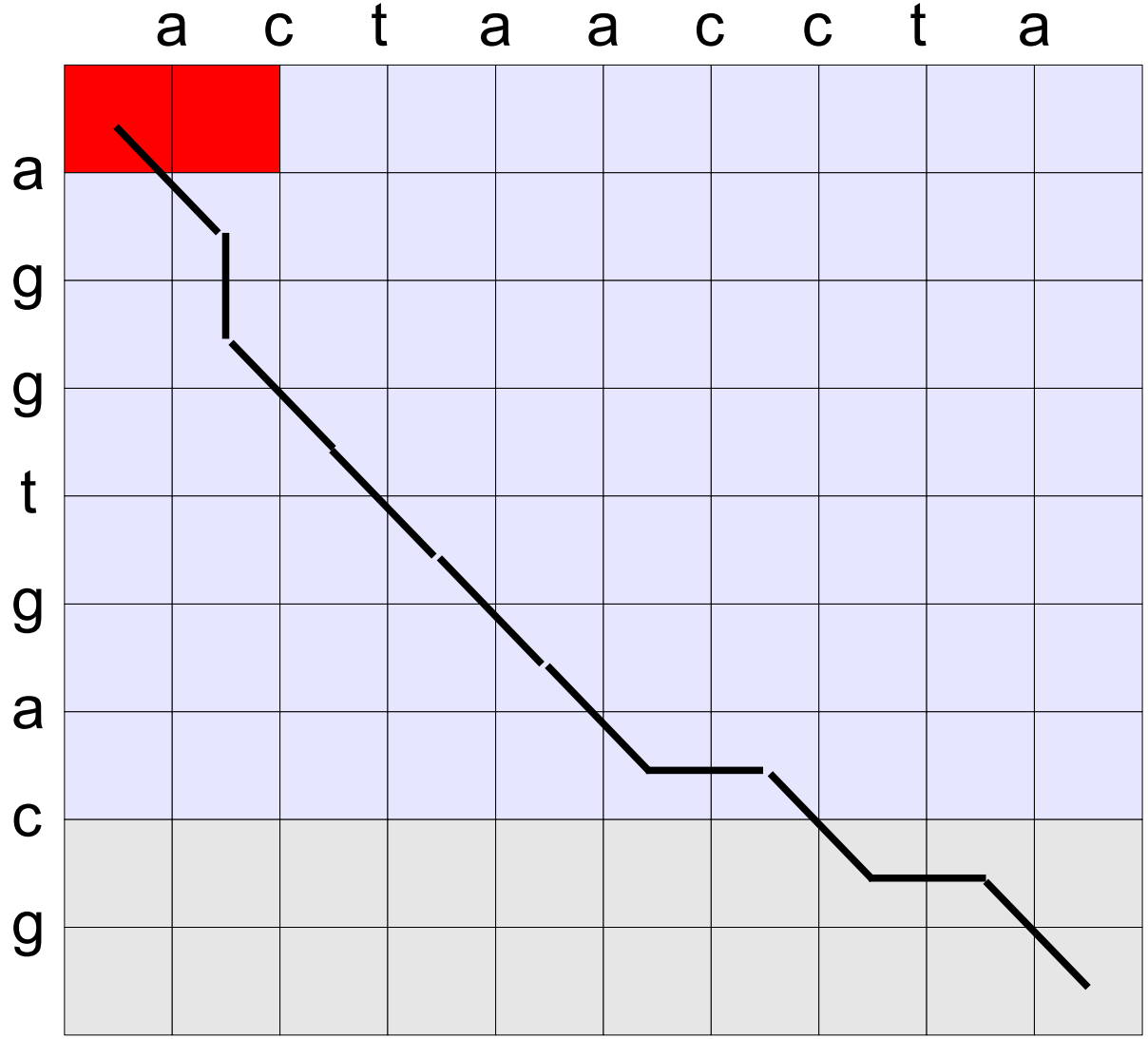
Time:

$nm +$
$(n - 2)m +$
$(n - 4)m +$
$(n - 6)m +$
2

Time analysis (in generel):

$$nm + (n\text{-}2)m + (n\text{-}4)m + \ldots + 2m = m(n + (n\text{-}2) + (n\text{-}4) + \ldots + 2) = \Theta(mn^2)$$

because $n(n\text{-}1)/4 <= n + (n\text{-}2) + (n\text{-}4) + \ldots + 2 <= n(n\text{-}1)/2$



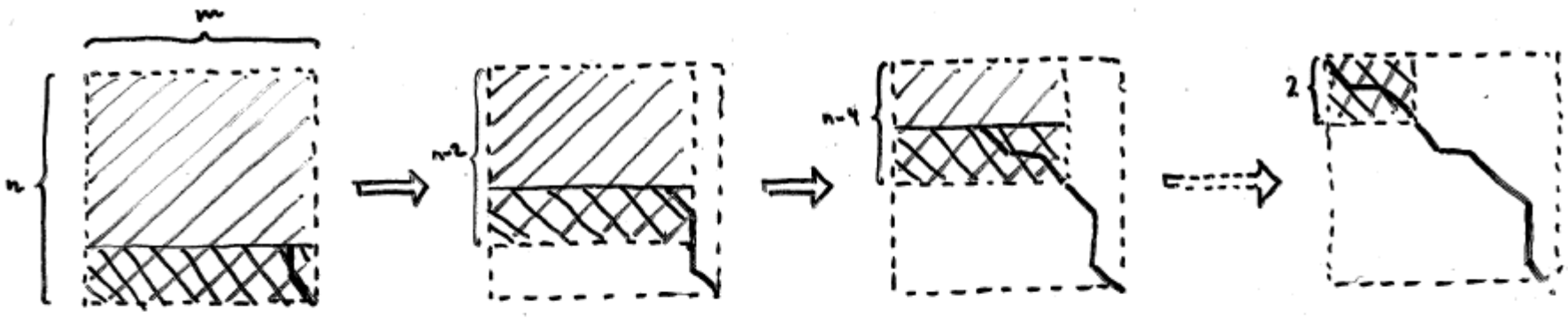
First idea:

Keep only two rows in memory and recompute when other parts of the matrix is needed during backtracking ...

Time:

$nm +$
$(n - 2)m +$
$(n - 4)m +$
$(n - 6)m +$
2

Linear space – First idea



$n \cdot m$ $\quad (n-2) \cdot m$ $\quad (n-4) \cdot m$ $\quad 2 \cdot m$

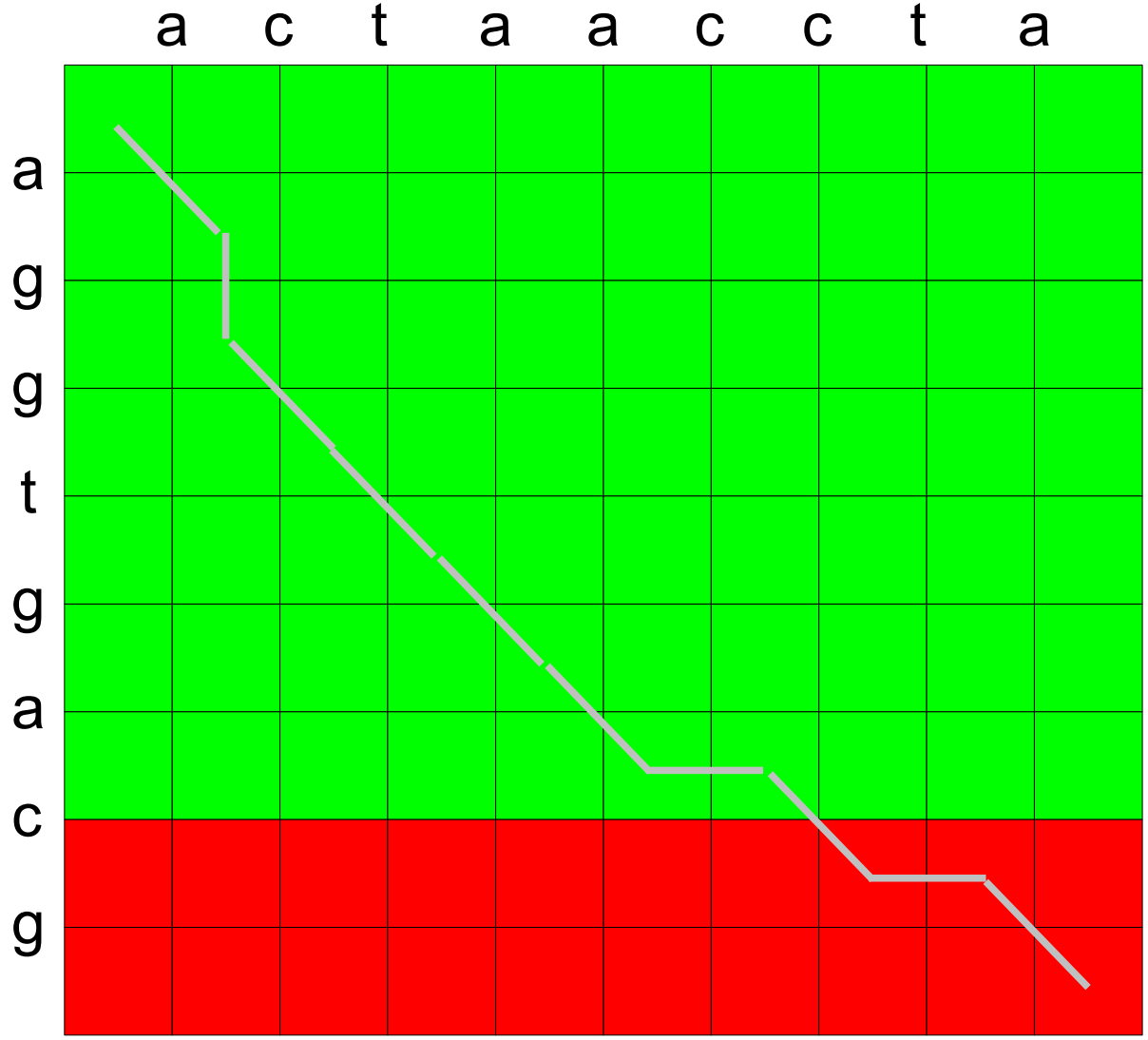
Dvs. tid ialt;

$$n \cdot m + (n-2) \cdot m + (n-4) \cdot m + \cdots + 2 \cdot m =$$

$$m \cdot (n + n-2 + n-4 + \ldots + 2) =$$

$$\underline{C(m \cdot n^2)}$$

Linear space – Hirschberg's idea



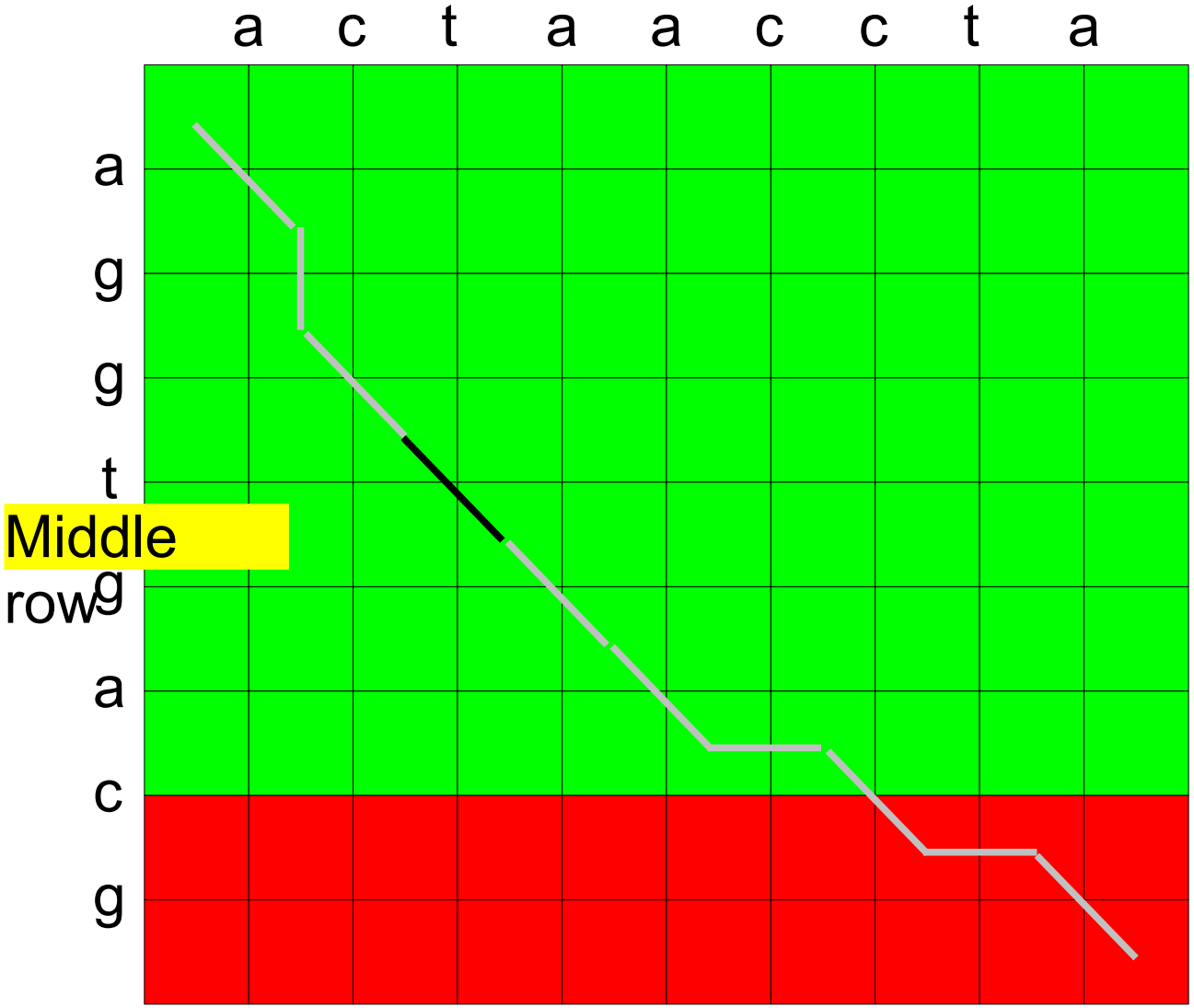
Hirschberg's idea:

Find “middle edge” (i.e. “middle column”) of an optimal alignment, split into two subproblems and recurse ...

Time:

nm

Linear space – Hirschberg's idea



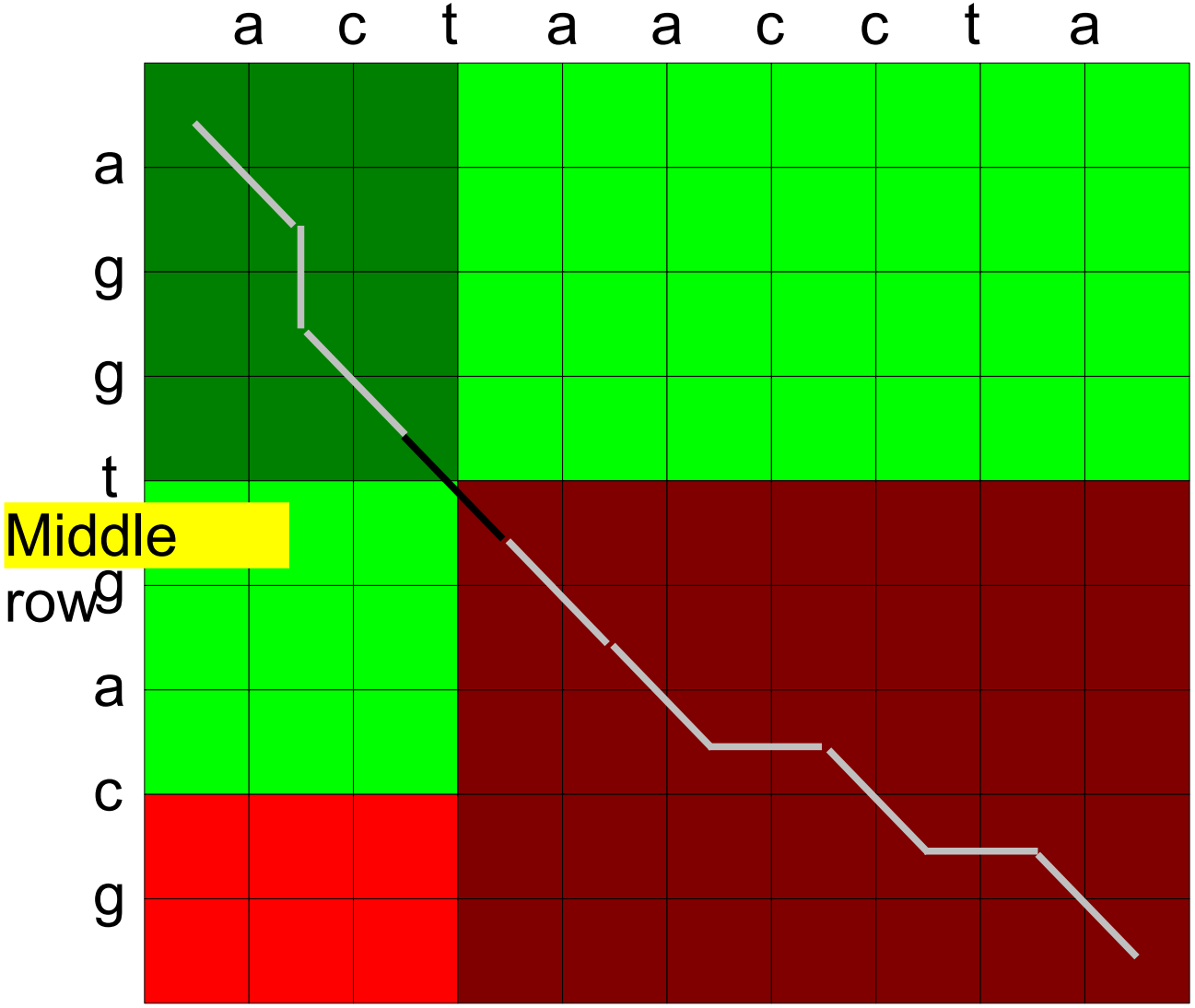
Hirschberg's idea:

Find “middle edge” (i.e. “middle column”) of an optimal alignment, split into two subproblems and recurse ...

Time:

nm

Linear space – Hirschberg's idea



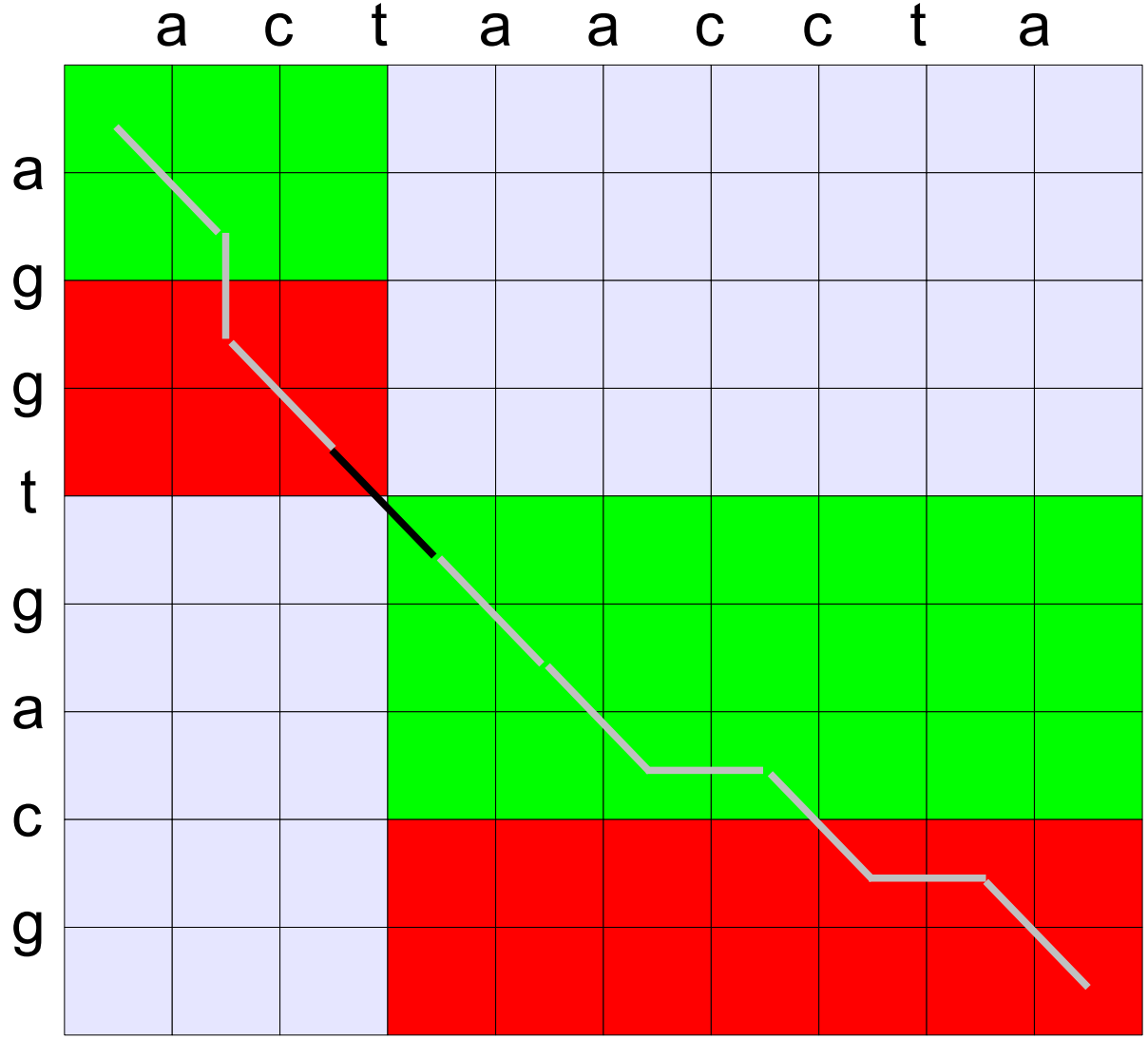
Hirschberg's idea:

Find “middle edge” (i.e. “middle column”) of an optimal alignment, split into two subproblems and recurse ...

Time:

nm

Linear space – Hirschberg's idea



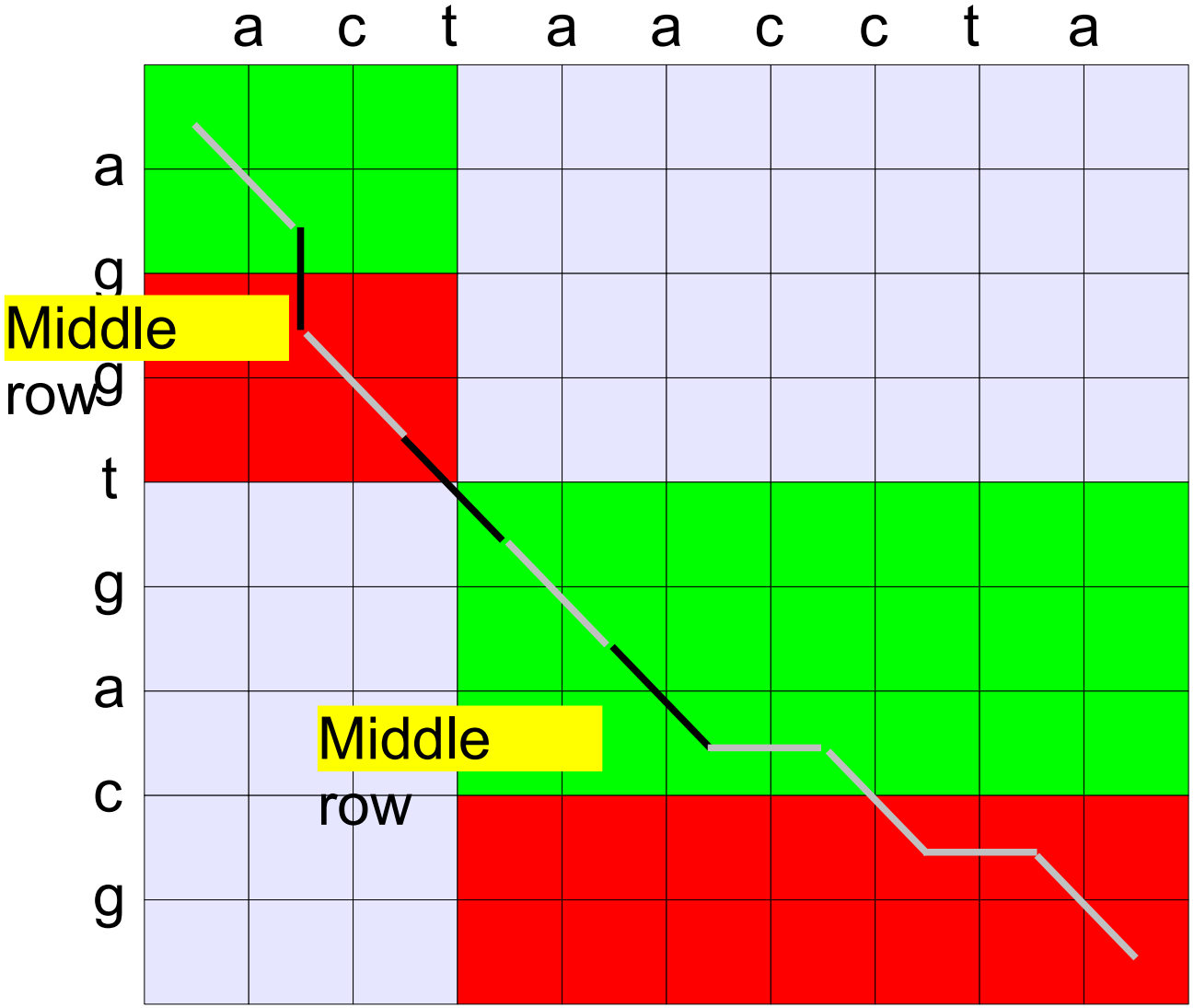
Hirschberg's idea:

Find “middle edge” (i.e. “middle column”) of an optimal alignment, split into two subproblems and recurse ...

Time:

$nm + (n / 2)m$

Linear space – Hirschberg's idea



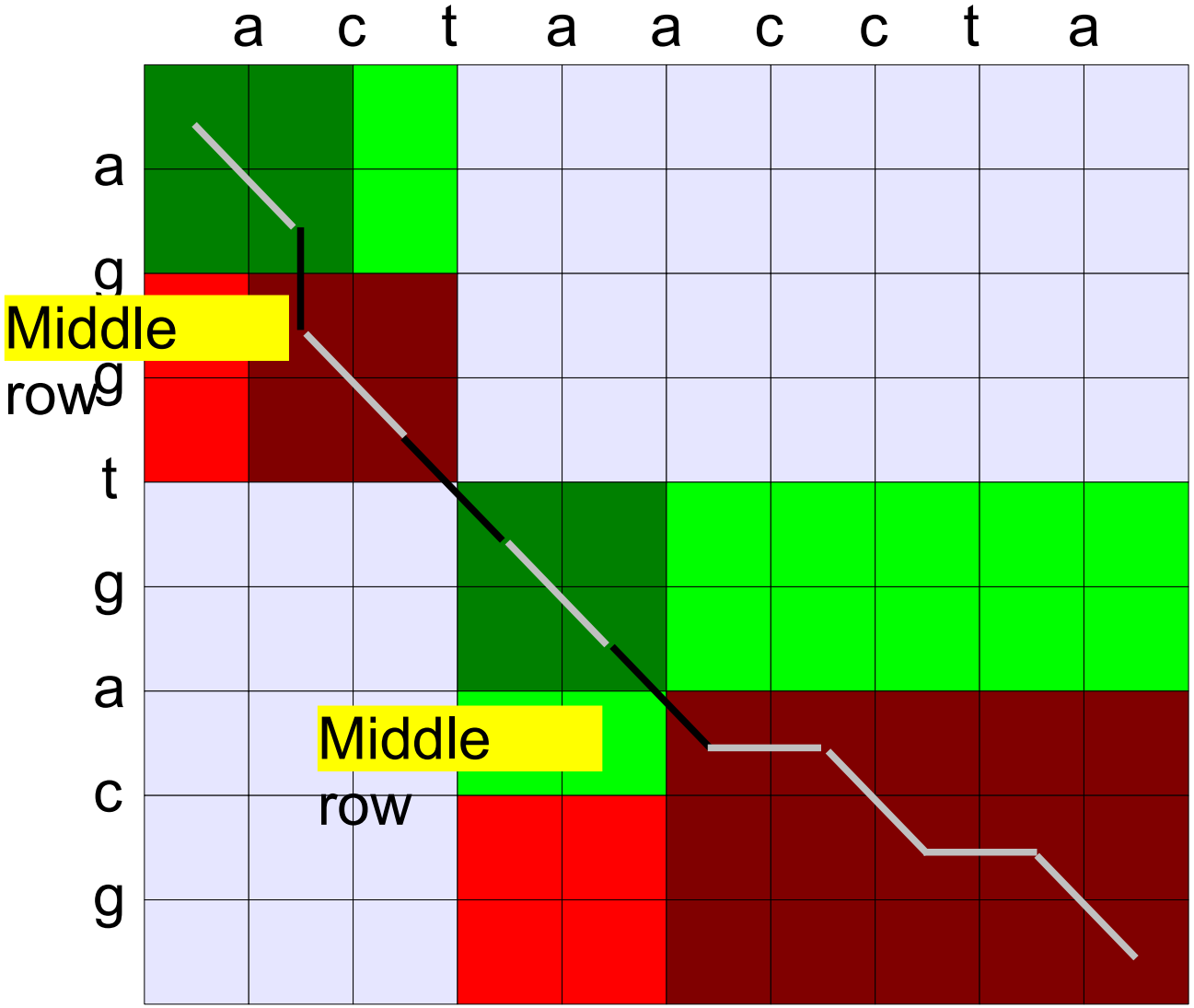
Hirschberg's idea:

Find “middle edge” (i.e. “middle column”) of an optimal alignment, split into two subproblems and recurse ...

Time:

$nm + (n / 2)m$

Linear space – Hirschberg's idea



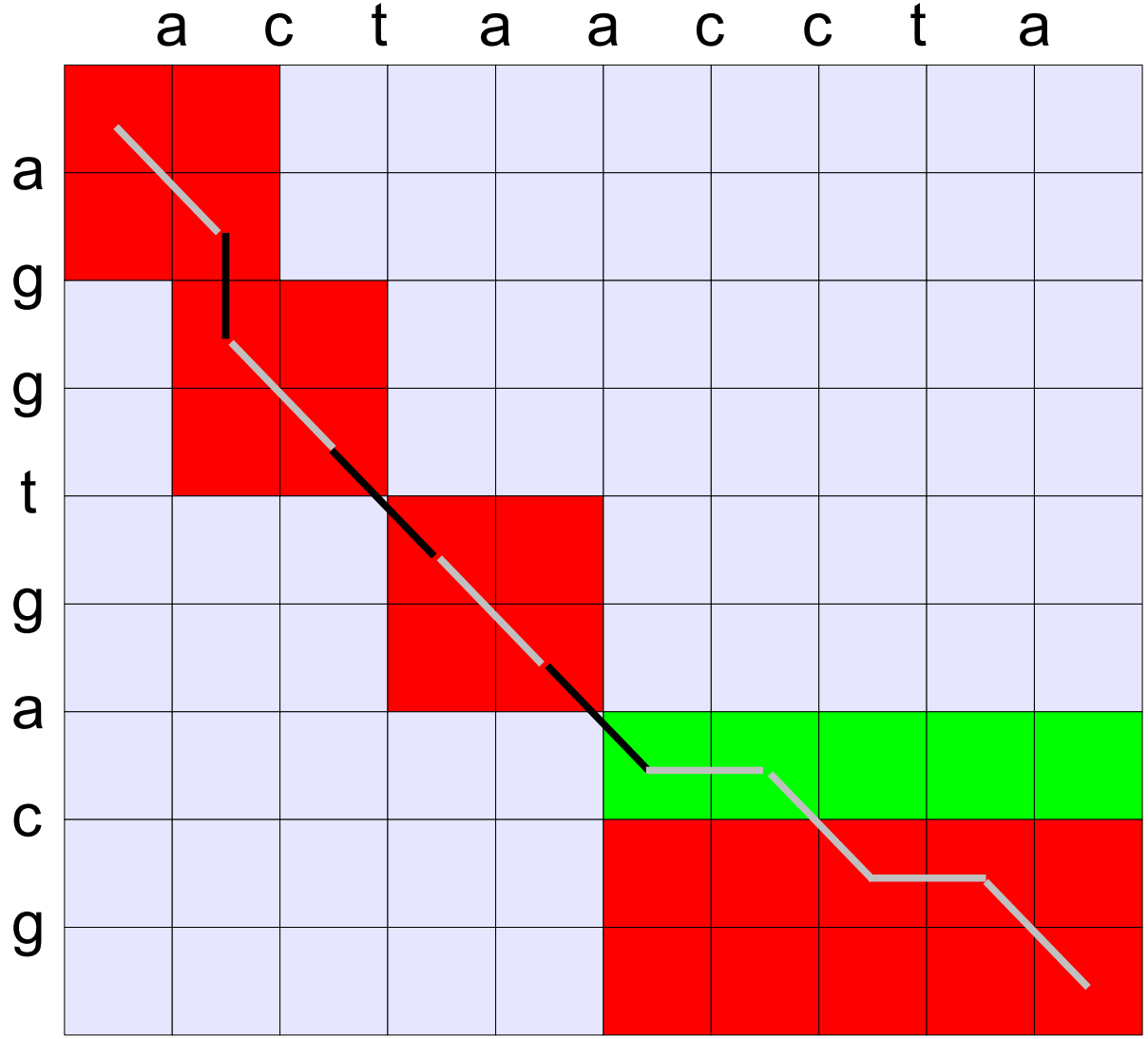
Hirschberg's idea:

Find “middle edge” (i.e. “middle column”) of an optimal alignment, split into two subproblems and recurse ...

Time:

$nm + (n / 2)m$

Linear space – Hirschberg's idea



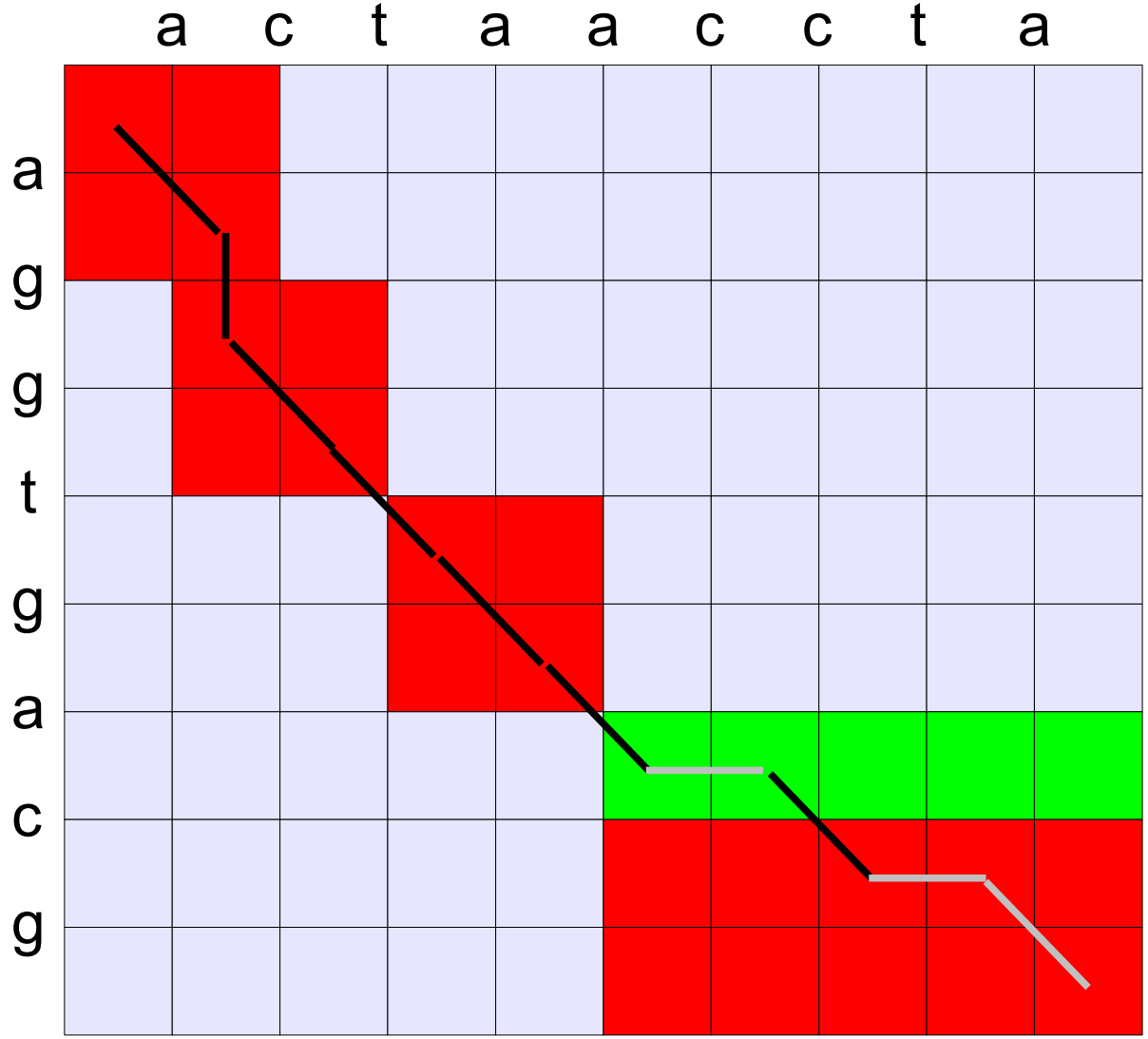
Hirschberg's idea:

Find “middle edge” (i.e. “middle column”) of an optimal alignment, split into two subproblems and recurse ...

Time:

$nm +$
$(n / 2)m +$
$(n / 4)m$

Linear space – Hirschberg's idea



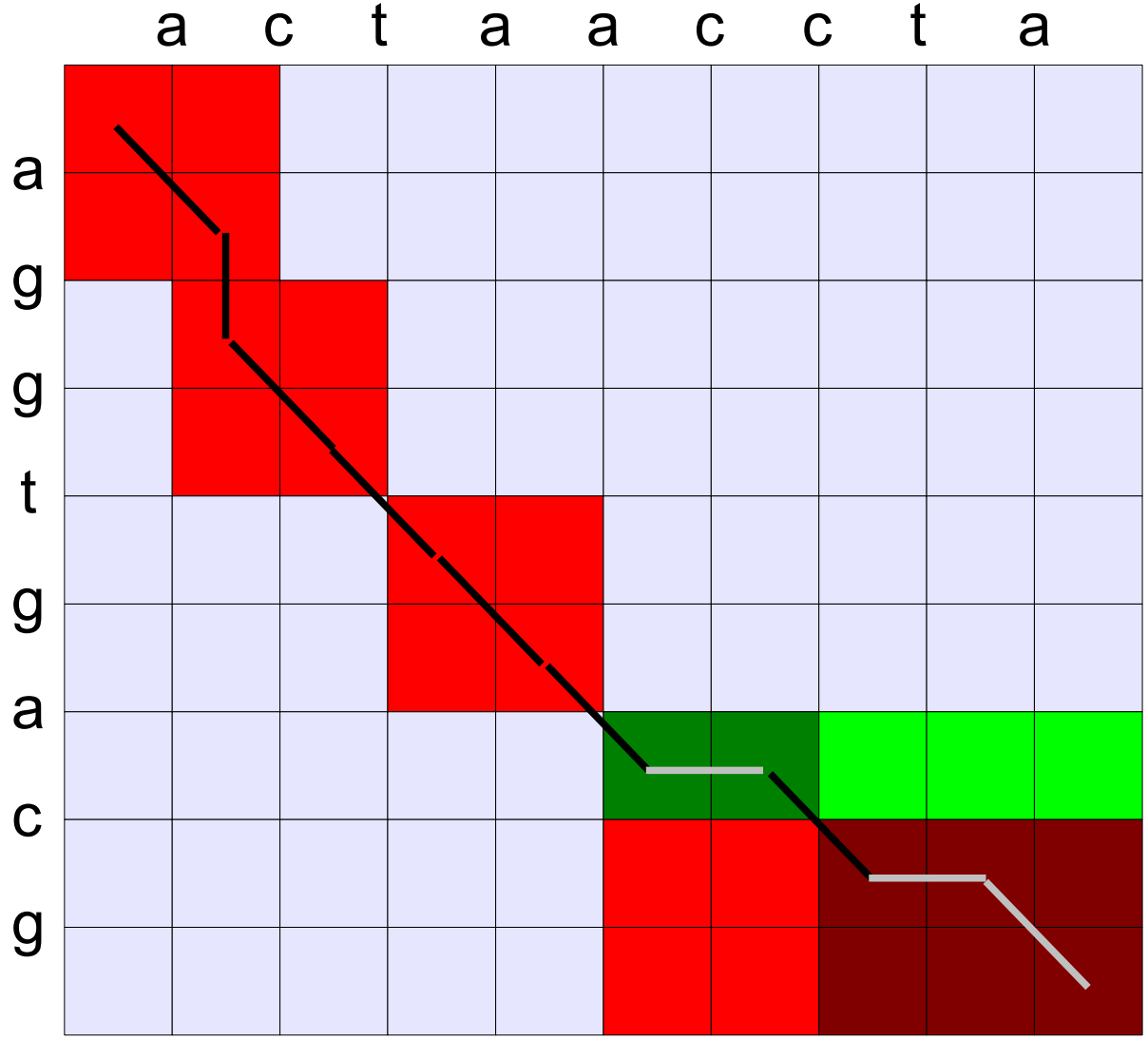
Hirschberg's idea:

Find “middle edge” (i.e. “middle column”) of an optimal alignment, split into two subproblems and recurse ...

Time:

$nm +$
$(n / 2)m +$
$(n / 4)m$

Linear space – Hirschberg's idea



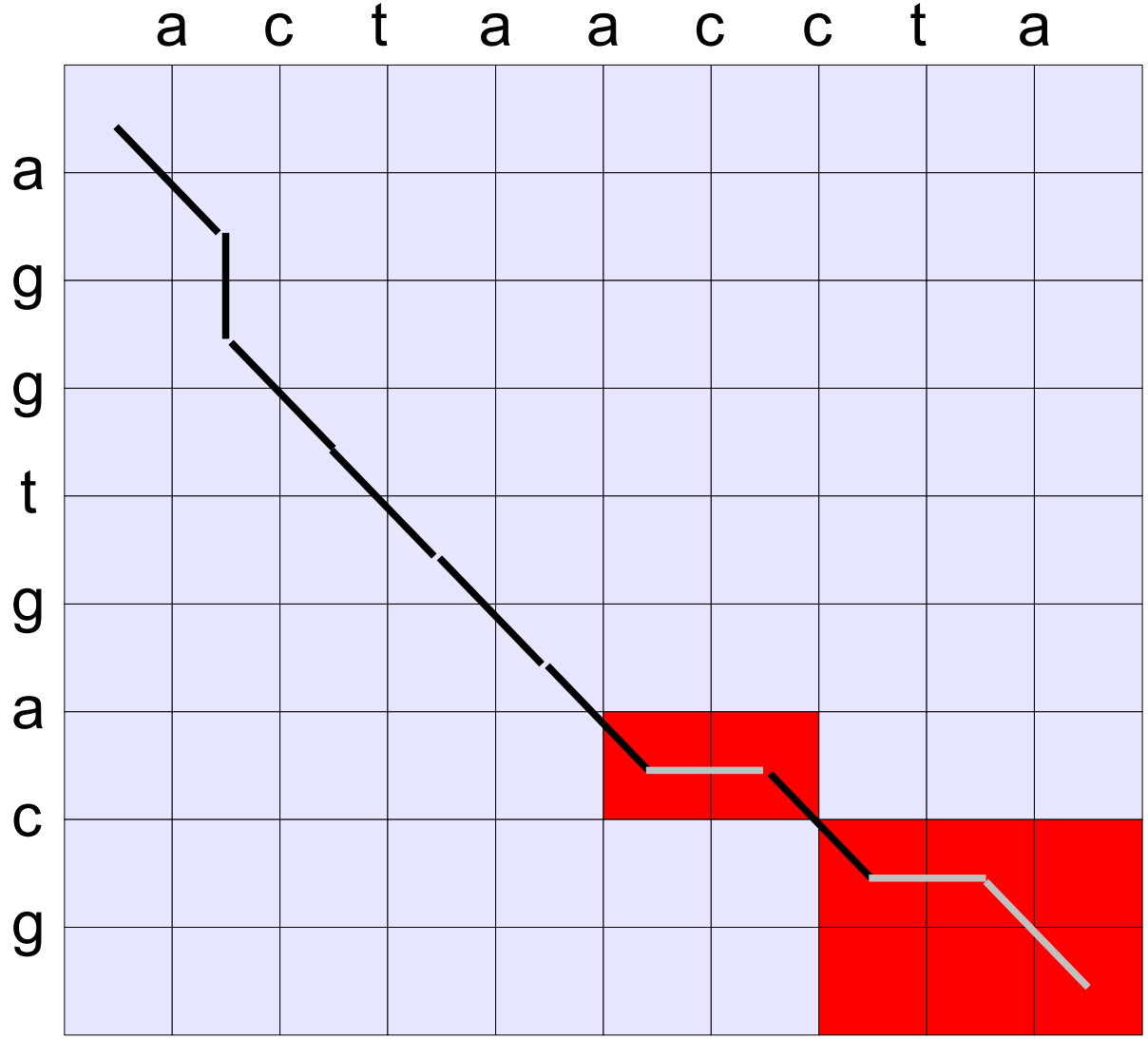
Hirschberg's idea:

Find “middle edge” (i.e. “middle column”) of an optimal alignment, split into two subproblems and recurse ...

Time:

$nm +$
$(n / 2)m +$
$(n / 4)m$

Linear space – Hirschberg's idea



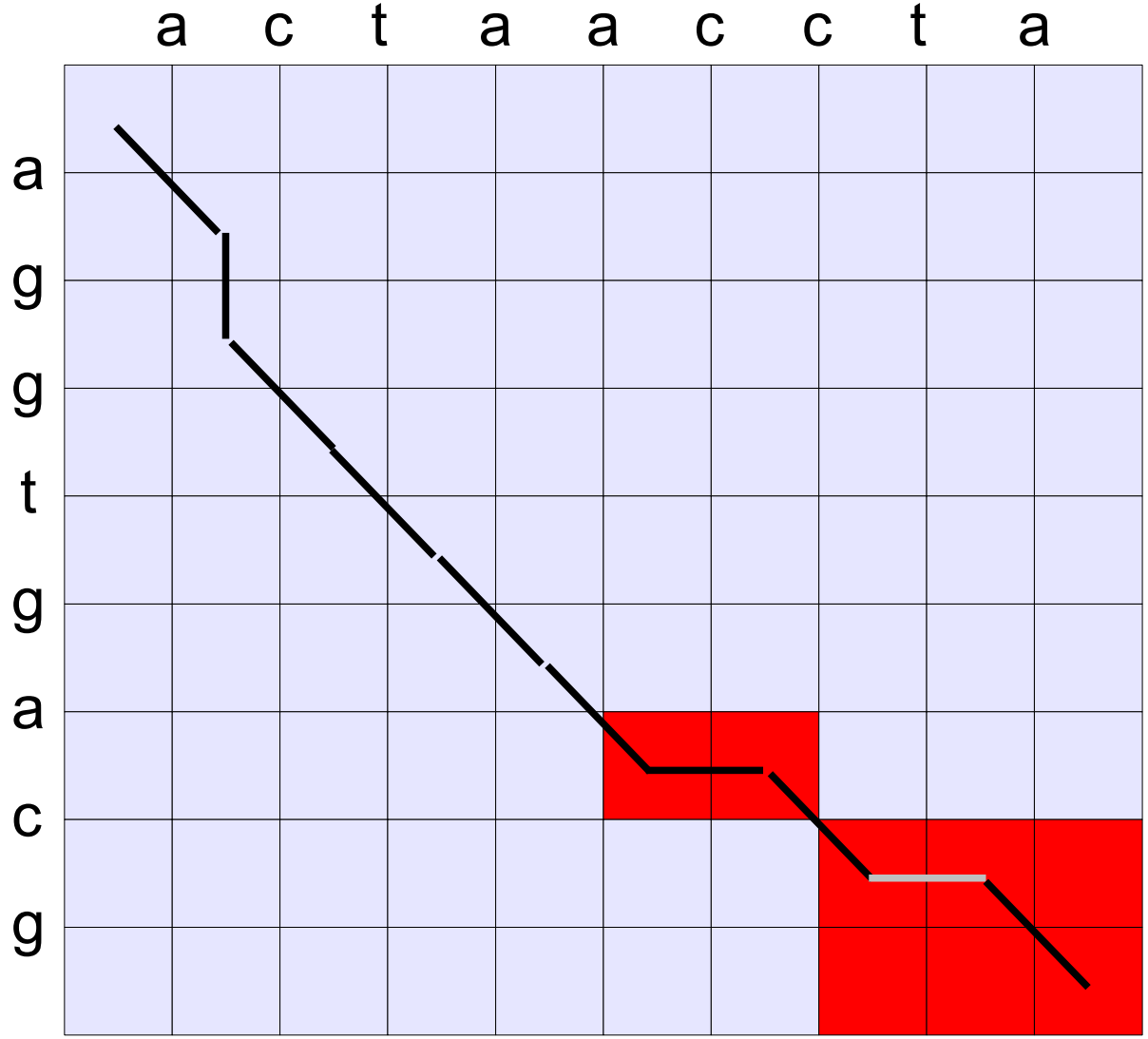
Hirschberg's idea:

Find “middle edge” (i.e. “middle column”) of an optimal alignment, split into two subproblems and recurse ...

Time:

$nm +$
$(n / 2)m +$
$(n / 4)m +$
$(n / 8)m$

Linear space – Hirschberg's idea



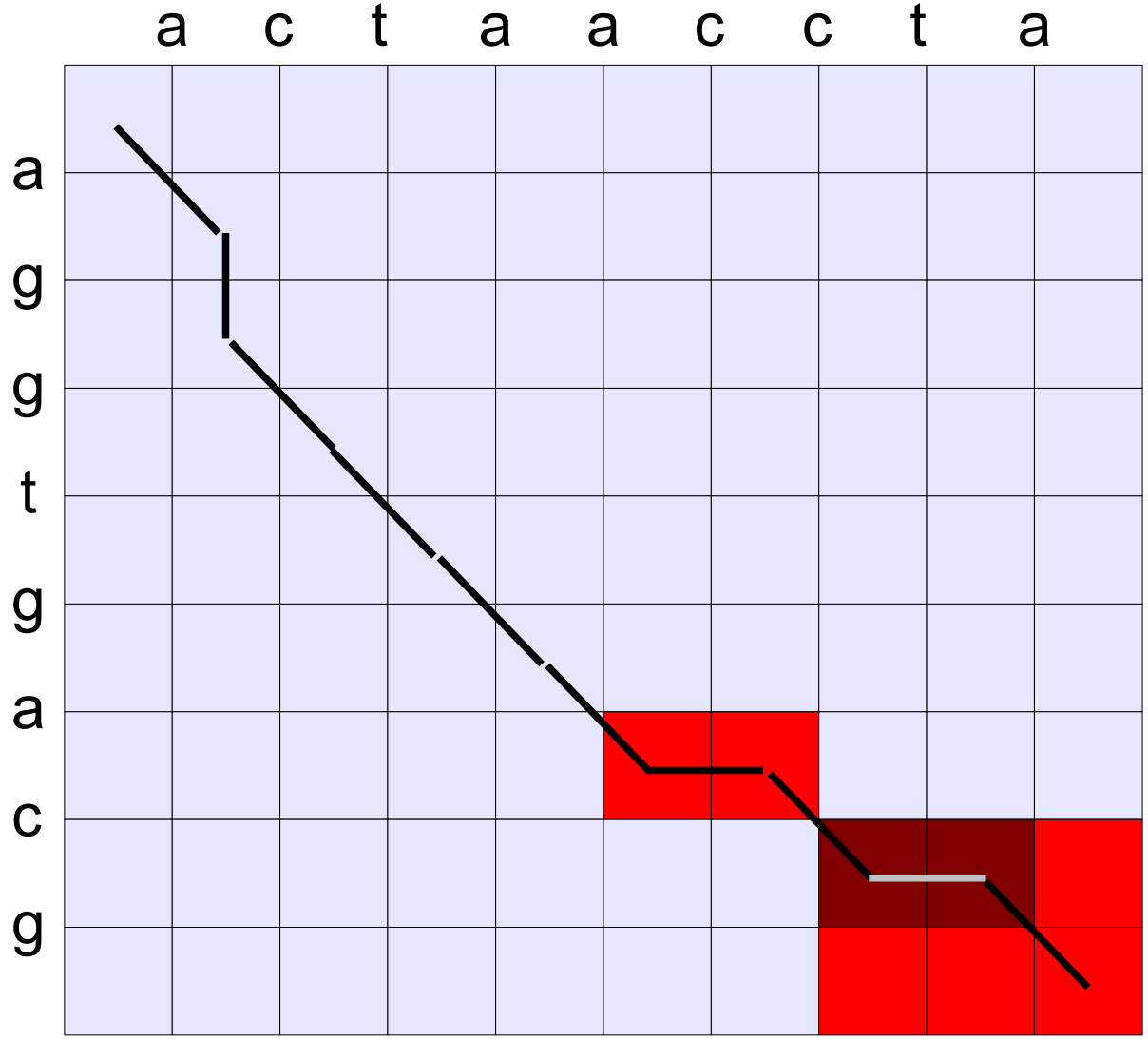
Hirschberg's idea:

Find “middle edge” (i.e. “middle column”) of an optimal alignment, split into two subproblems and recurse ...

Time:

$nm +$
$(n / 2)m +$
$(n / 4)m +$
$(n / 8)m$

Linear space – Hirschberg's idea



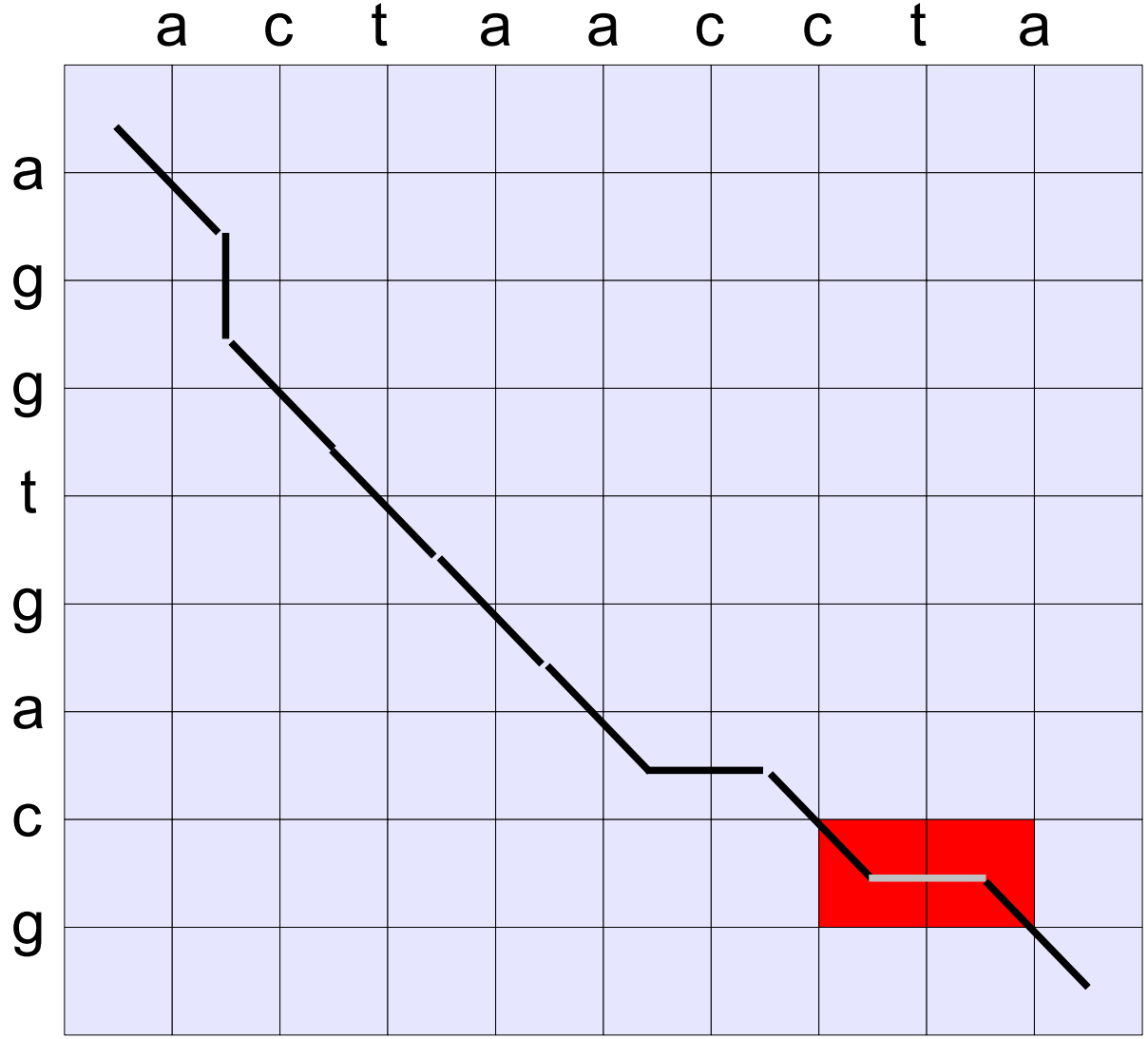
Hirschberg's idea:

Find “middle edge” (i.e. “middle column”) of an optimal alignment, split into two subproblems and recurse ...

Time:

$nm +$
$(n / 2)m +$
$(n / 4)m +$
$(n / 8)m$

Linear space – Hirschberg's idea



Hirschberg's idea:

Find “middle edge” (i.e. “middle column”) of an optimal alignment, split into two subproblems and recurse ...

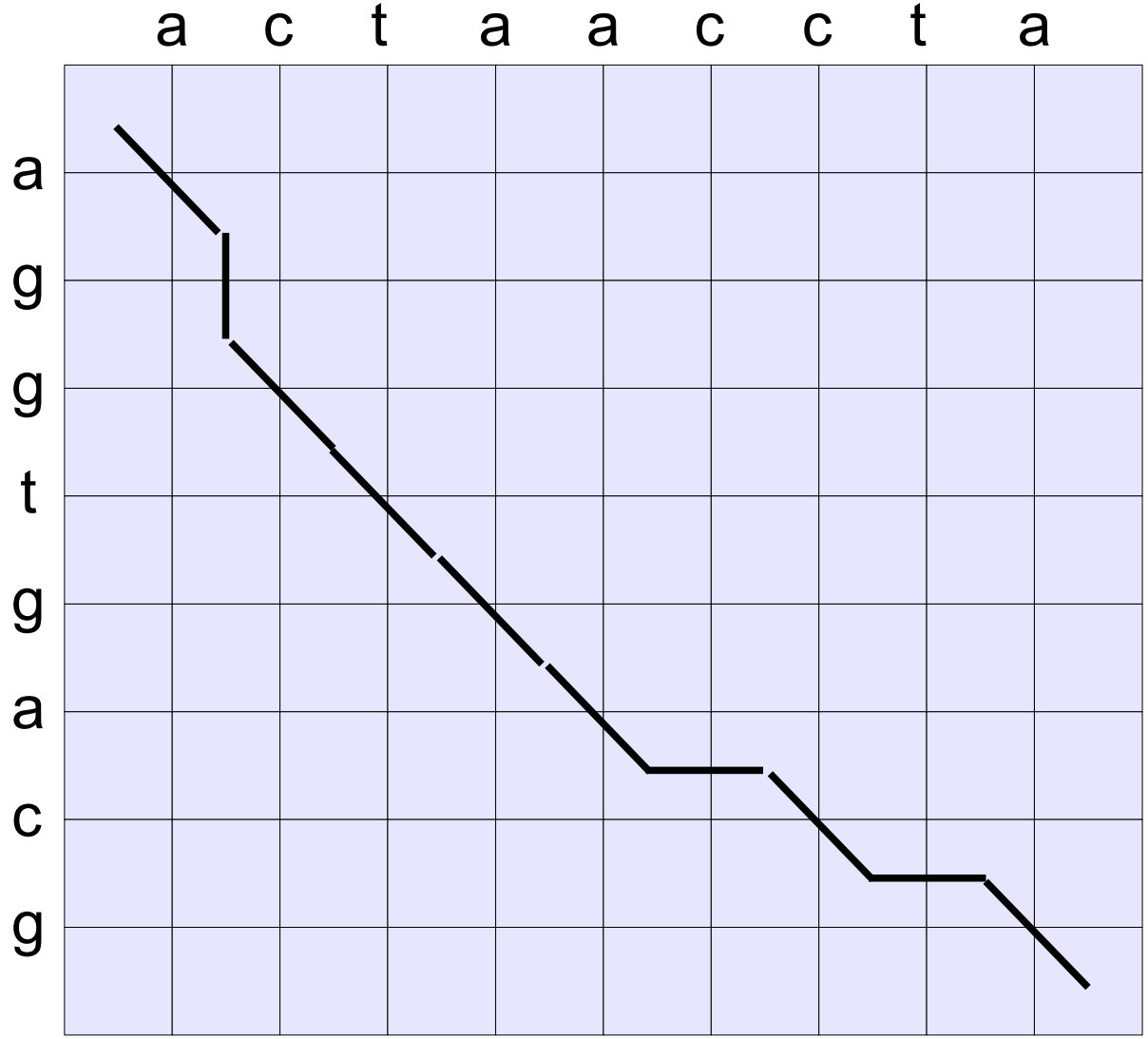
Time:

$nm +$
$(n / 2)m +$
$(n / 4)m +$
$(n / 8)m$

Time analysis (in generel):

$$nm + (n/2)m + (n/4)m + \ldots + 2m = m(n + (n/2) + (n/4) + \ldots + 2) = \Theta(mn)$$

because $n <= n + (n/2) + (n/4) + \ldots + 2 <= 2n$



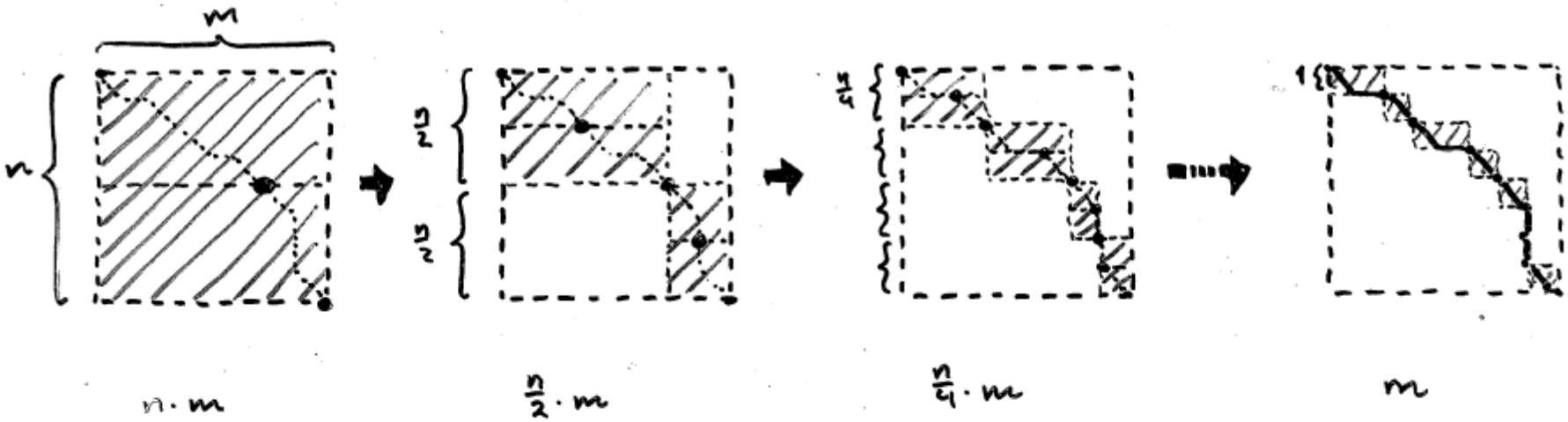
Hirschberg's idea:

Find “middle edge” (i.e. “middle column”) of an optimal alignment, split into two subproblems and recurse ...

Time:

$nm +$
$(n / 2)m +$
$(n / 4)m +$
$(n / 8)m$

Linear space – Hirschberg's idea



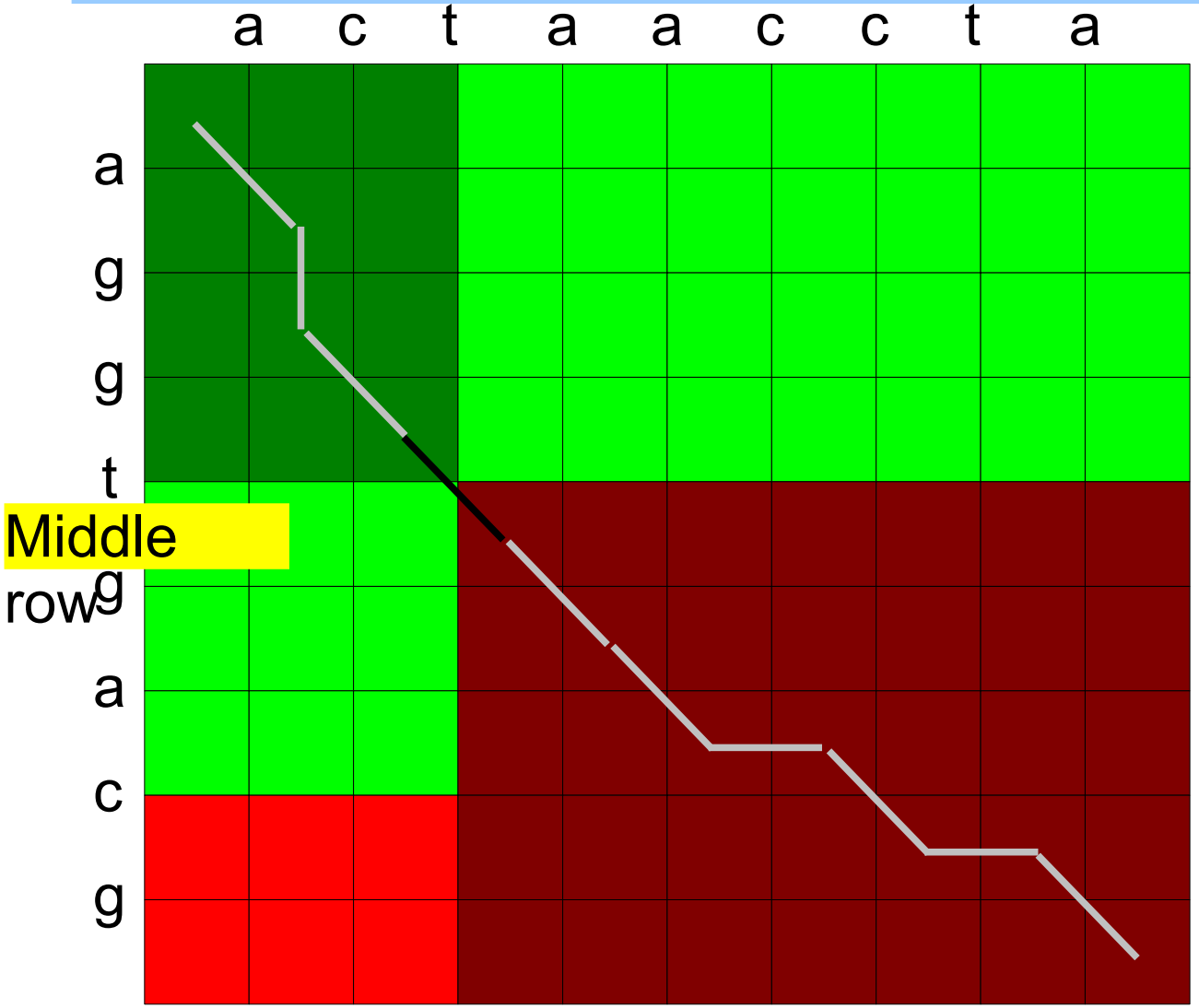
Dvs. tid ialt;

$$n \cdot m + \frac{n}{2} \cdot m + \frac{n}{4} \cdot m + \cdots + m = m \cdot \left(n + \frac{n}{2} + \frac{n}{4} + \cdots + 1\right) < m \cdot 2n = \underline{O(n \cdot m)}$$

og plads $O(m)$ hvis vi kan finde ud af hvor den længste vej krydser midten uden at gemme hele tabel h...

Linear space – Hirschberg's idea

Problem: Find the “middle column” (aka “middle edge”) in an optimal alignment in quadratic time and linear space ...



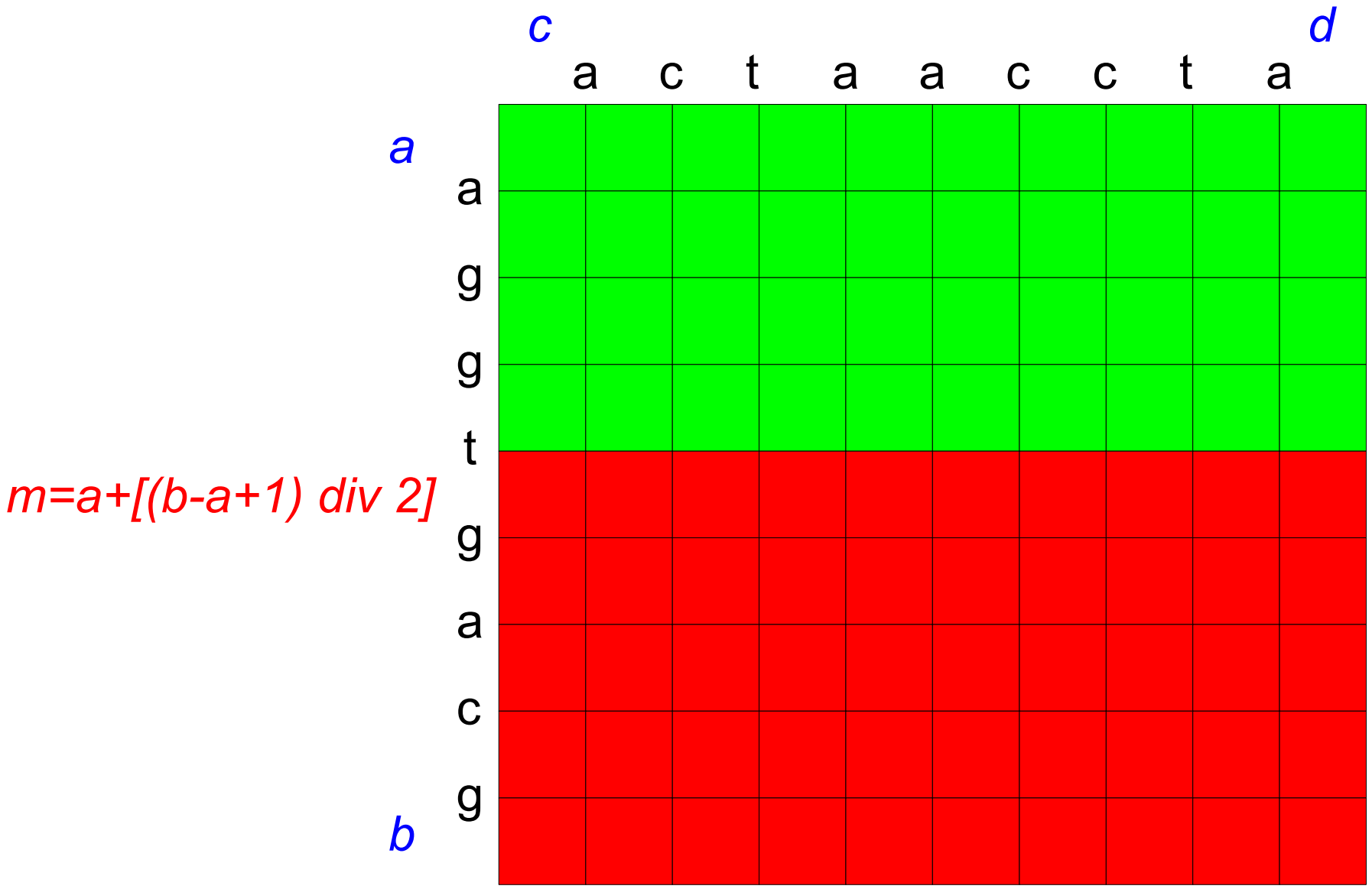
Hirschberg's idea:

Find “middle edge” (i.e. “middle column”) of an optimal alignment, split into two subproblems and recurse ...

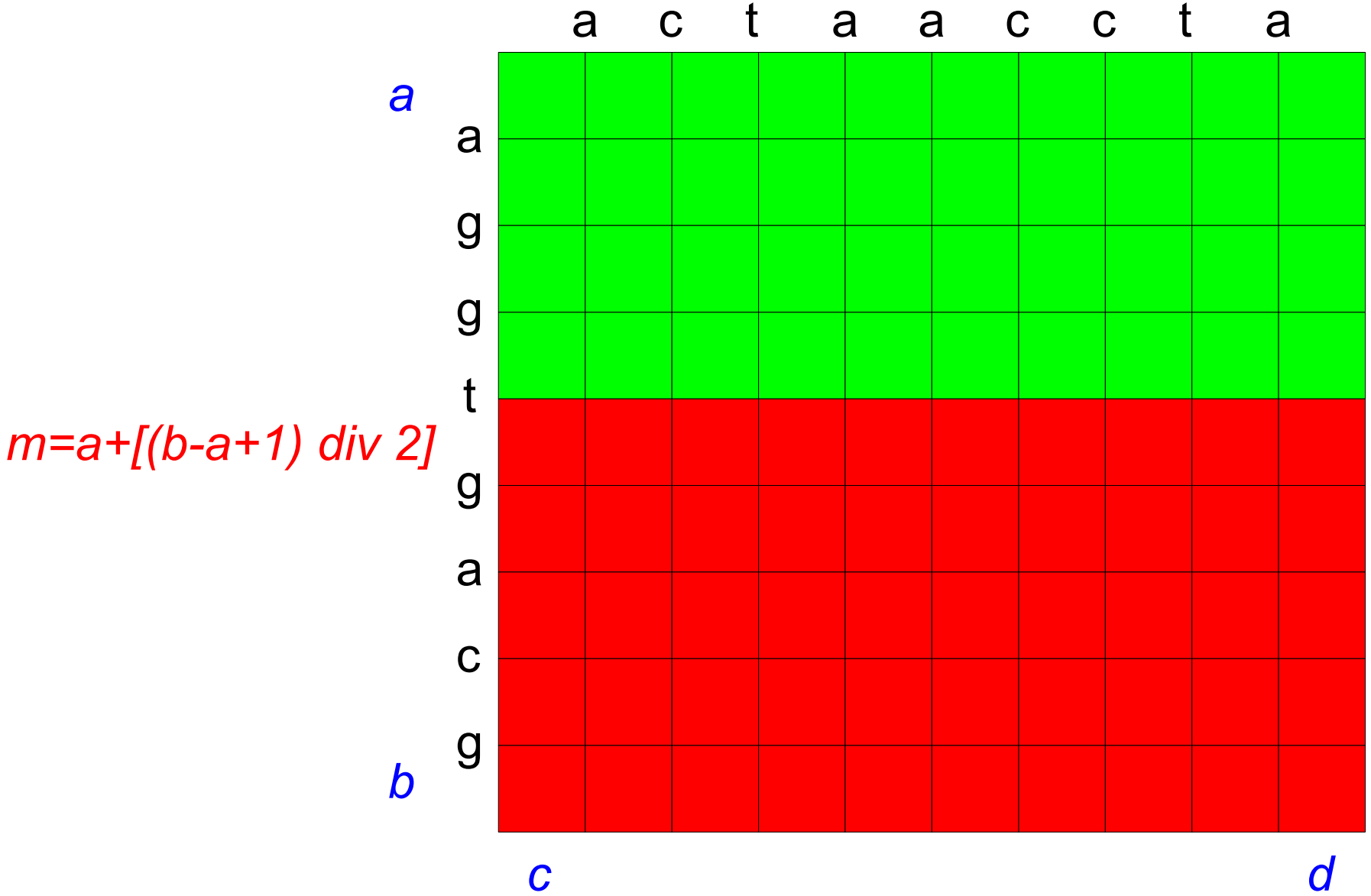
Time:

$nm +$
$(n / 2)m +$
$(n / 4)m +$
$(n / 8)m$

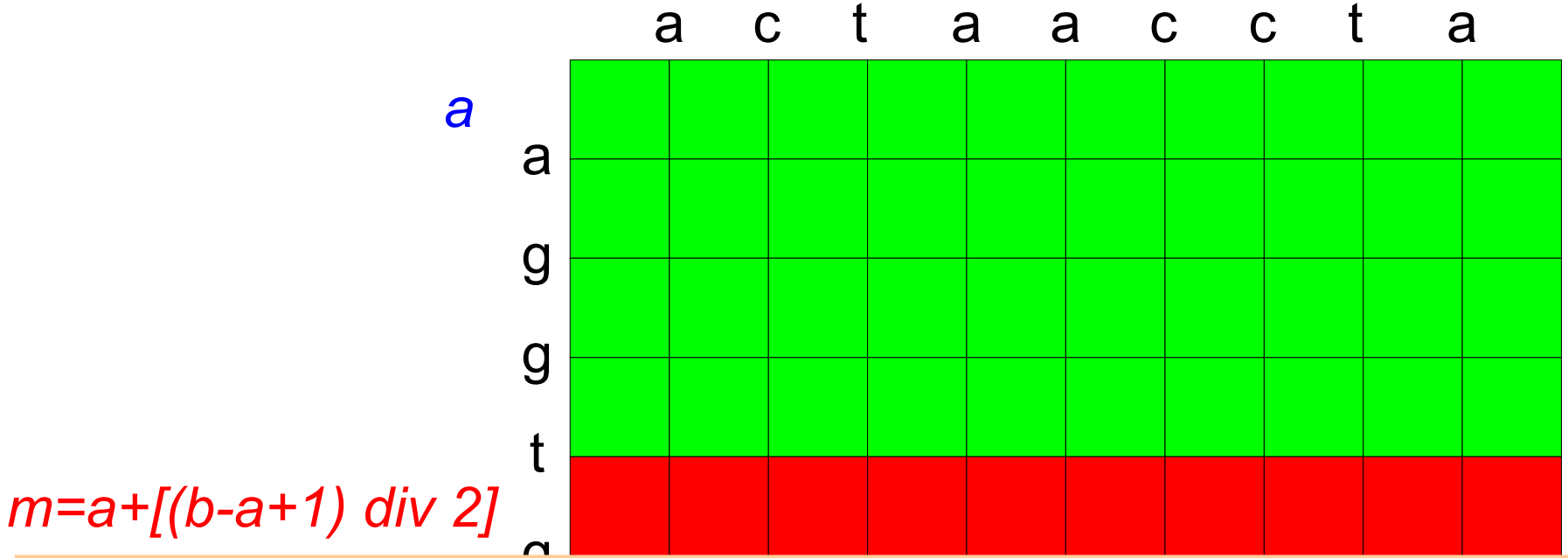
The “interesting part” of the table



Problem: Find the “middle column” (aka “middle edge”) in an optimal alignment of A[a+1..b] and B[c+1..d], i.e. find the edge on an optimal path from node (a,c) to node (b,d) in the alignment graph which goes from the upper half (green) to the lower half (red) in the “dynamic programming table” below.



Problem: Find the “middle column” (aka “middle edge”) in an optimal alignment of A[a+1..b] and B[c+1..d], i.e. find the edge on an optimal path from node (a,c) to node (b,d) in the alignment graph which goes from the upper half (green) to the lower half (red) in the “dynamic programming table” below.



Solution idea: We define a function ***find_edge(a,b,c,d,m)*** which returns the edge from row m-1 to row m on an optimal path from (a,c) to cell (b,d), i.e. the "middle column" in an optimal alignment of A[a+1..b] and B[c+1..d]. The edge can either be a diagonal edge going from (m-1,j-1) to (m,j) or a vertical edge going from (m-1,j) to (m,j) for some c<=j<=d. We denote the two cases as **(j,DIAGONAL)** and **(j,VERTICAL)** respectively.

Implementing find_edge(a,b,c,d,m)

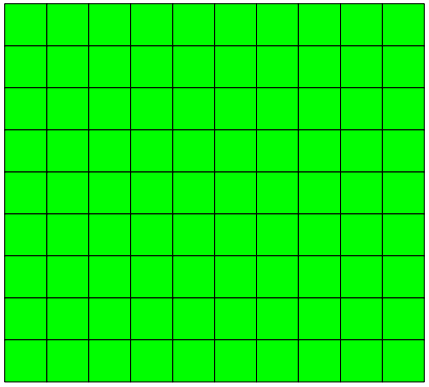
Implementation: Define the ***traceback table T[a..b][c..d]*** as a (b-a+1)*(d-c+1) table such that entry T[i,j] identifies the edge on an optimal path from (a,c) to (i,j) which goes from row m-1 to row m. The edge is either (x,DIAGONAL) or (x,VERTICAL) for some c<=x<=d. In particular T[c,d] is the edge which goes from row m-1 to row m on an optimal path from (a,c) to (b,d), i.e. the edge we are looking for.

Implementing find_edge(a,b,c,d,m)

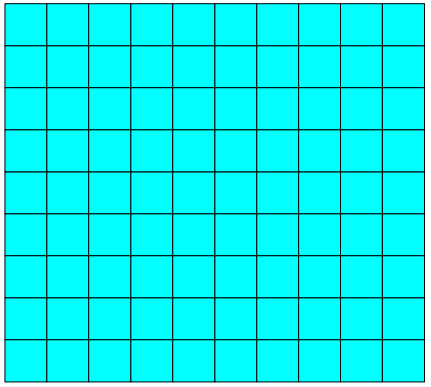
Implementation: Define the ***traceback table T[a..b][c..d]*** as a (b-a+1)*(d-c+1) table such that entry T[i,j] identifies the edge on an optimal path from (a,c) to (i,j) which goes from row m-1 to row m. The edge is either (x,DIAGONAL) or (x,VERTICAL) for some c<=x<=d. In particular T[c,d] is the edge which goes from row m-1 to row m on an optimal path from (a,c) to (b,d), i.e. the edge we are looking for.

Computation: We can compute T[c,d] by filling out the traceback table T[a..b][c..d] row by row while filling the corresponding cells in the dynamic programming table S[a..b][c..d].

S:



T:



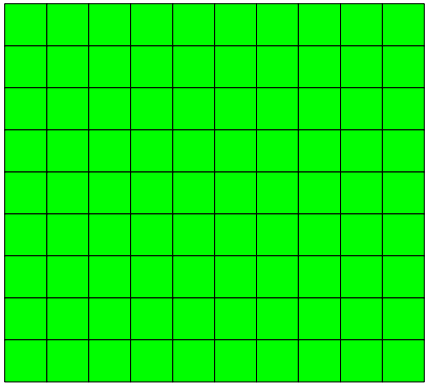
Implementing find_edge(a,b,c,d,m)

Implementation: Define the ***traceback table T[a..b][c..d]*** as a (b-a+1)*(d-c+1) table such that entry T[i,j] identifies the edge on an optimal path from (a,c) to (i,j) which goes from row m-1 to row m. The edge is either (x.DIAGONAL) or (x.VERTICAL) for some c<=x<=d. I row m-1 to row m o ge we are looking for.

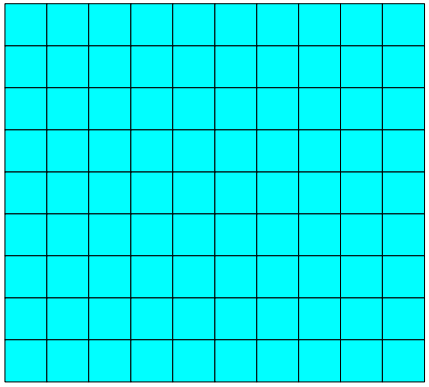
$$S[i, j] = \max \begin{cases} S[i-1, j-1] + \text{subcost}(A[i\,], B[j\,]) \\ S[i-1, j] + \text{gapcost} \\ S[i, j-1] + \text{gapcost} \\ 0 \text{ if } i=0 \text{ and } j=0 \end{cases}$$

Computati aceback table T[a..b][c..d] row by row while filling the corresponding cells in the dynamic programming table S[a..b][c..d].

S:



T:



Computing T[i,j]

If i < m:

T[i,j] = None.

Intuition: an optimal path from (a,c)->..->(i,j) does not go from row m-1 to row m

if i = m:

case 1: if S[i,j] = S[i-1,j-1] + subcost(A[i],B[j]): T[i,j] = (j, DIAGONAL)

Intuition: an optimal path is (a,c)->..->(m-1,j-1)->(m,j)

case 2: if S[i,j] = S[i-1,j] + gapcost: T[i,j] = (j, VERTICAL)

Intuition: an optimal path is (a,c)->..->(m-1,j)->(m,j)

case 3: if S[i,j] = S[i,j-1] + gapcost: T[i,j] = T[i,j-1]

Intuition: an optimal path is (a,c)->..->(m,j-1)->(m,j)

if i > m:

if S[i,j] = S[i-1,j-1] + subcost(A[i],B[j]): T[i,j] = T[i-1,j-1]

if S[i,j] = S[i-1,j] + gapcost: T[i,j] = T[i-1,j]

if S[i,j] = S[i,j-1] + gapcost: T[i,j] = T[i,j-1]

Intuition: an optimal path is (a,c)->..->(h,k)->(i,j), i.e. it goes from row m-1 to row m via the same edge as an optimal path (a,b)->..->(h,k)

Observation: We can compute table T row by row in linear space. While filling out row i, we do not access entries in row 0..i-2, i.e. we only need to keep two rows of T in memory. (Exactly like when filling out the dynamic programming table).

If i < m:

T[i,j] = None.

Intuition: an optimal path from (a,c)->..->(i,j) does not go from row m-1 to row m

if i = m:

case 1: if S[i,j] = S[i-1,j-1] + subcost(A[i],B[j]): T[i,j] = (j, DIAGONAL)

Intuition: an optimal path is (a,c)->..->(m-1,j-1)->(m,j)

case 2: if S[i,j] = S[i-1,j] + gapcost: T[i,j] = (j, VERTICAL)

Intuition: an optimal path is (a,c)->..->(m-1,j)->(m,j)

case 3: if S[i,j] = S[i,j-1] + gapcost: T[i,j] = T[i,j-1]

Intuition: an optimal path is (a,c)->..->(m,j-1)->(m,j)

if i > m:

if S[i,j] = S[i-1,j-1] + subcost(A[i],B[j]): T[i,j] = T[i-1,j-1]

if S[i,j] = S[i-1,j] + gapcost: T[i,j] = T[i-1,j]

if S[i,j] = S[i,j-1] + gapcost: T[i,j] = T[i,j-1]

Intuition: an optimal path is (a,c)->..->(h,k)->(i,j), i.e. it goes from row m-1 to row m via the same edge as an optimal path (a,b)->..->(h,k)

Splitting into subproblems

When *find_edge(a,b,c,d,m)* has been computed, we have found an edge/column in an optimal alignment of A[a+1..b] and B[c+1..d] and can split the remaining alignment problem into two subproblems cf. below and the figures on the following two slides.

If *find_edge(a,b,c,d,m) = (j,DIAGONAL)*, we can split the remaining alignment problem into two subproblems:

find the optimal alignment of A[a+1..m-1] and B[c+1..j-1]

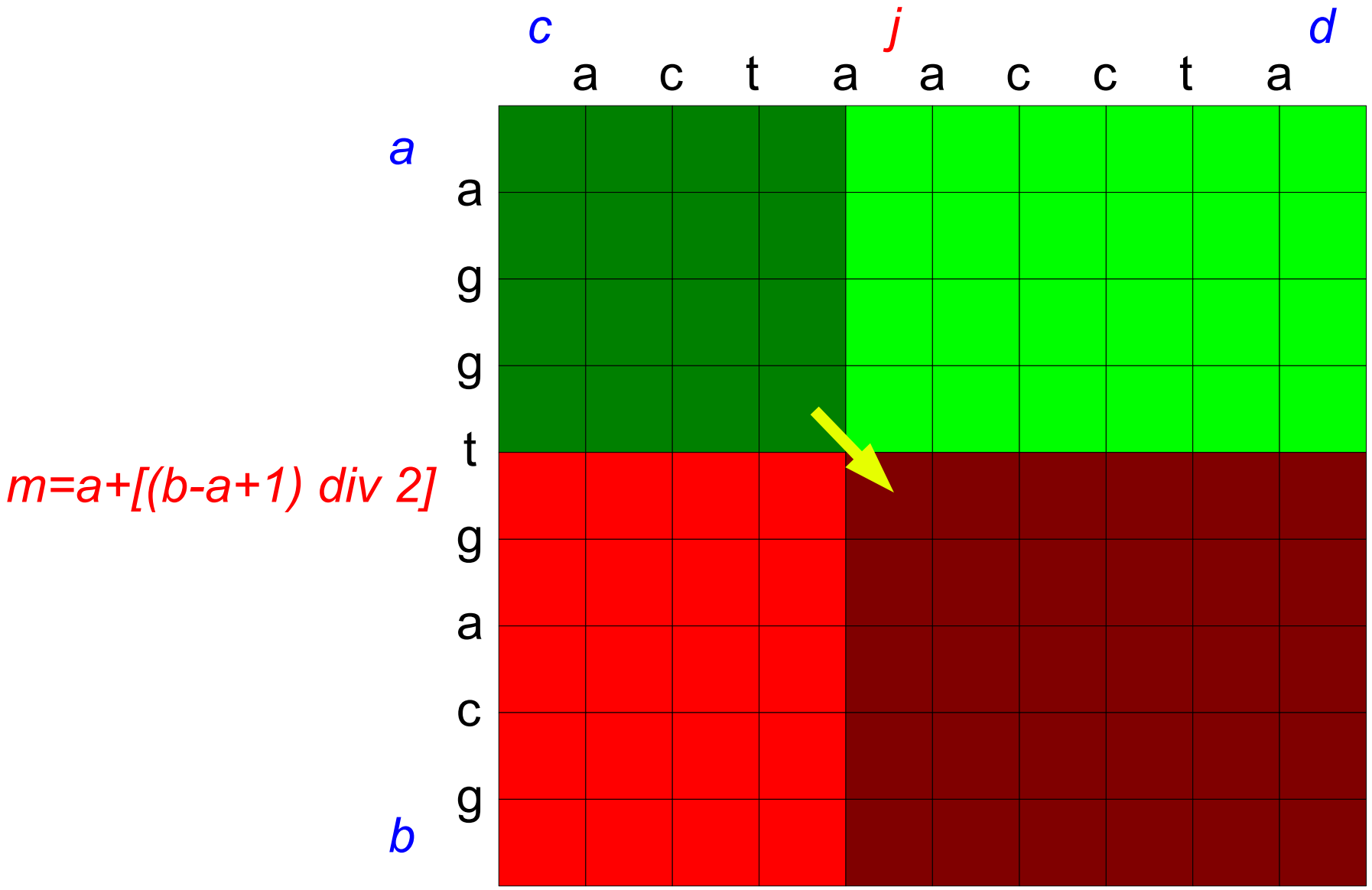
find the optimal alignment of A[m+1..b] and B[j+1..d]

If *find_edge(a,b,c,d,m) = (j,VERTICAL)*, we can split the remaining alignment problem into two subproblems:

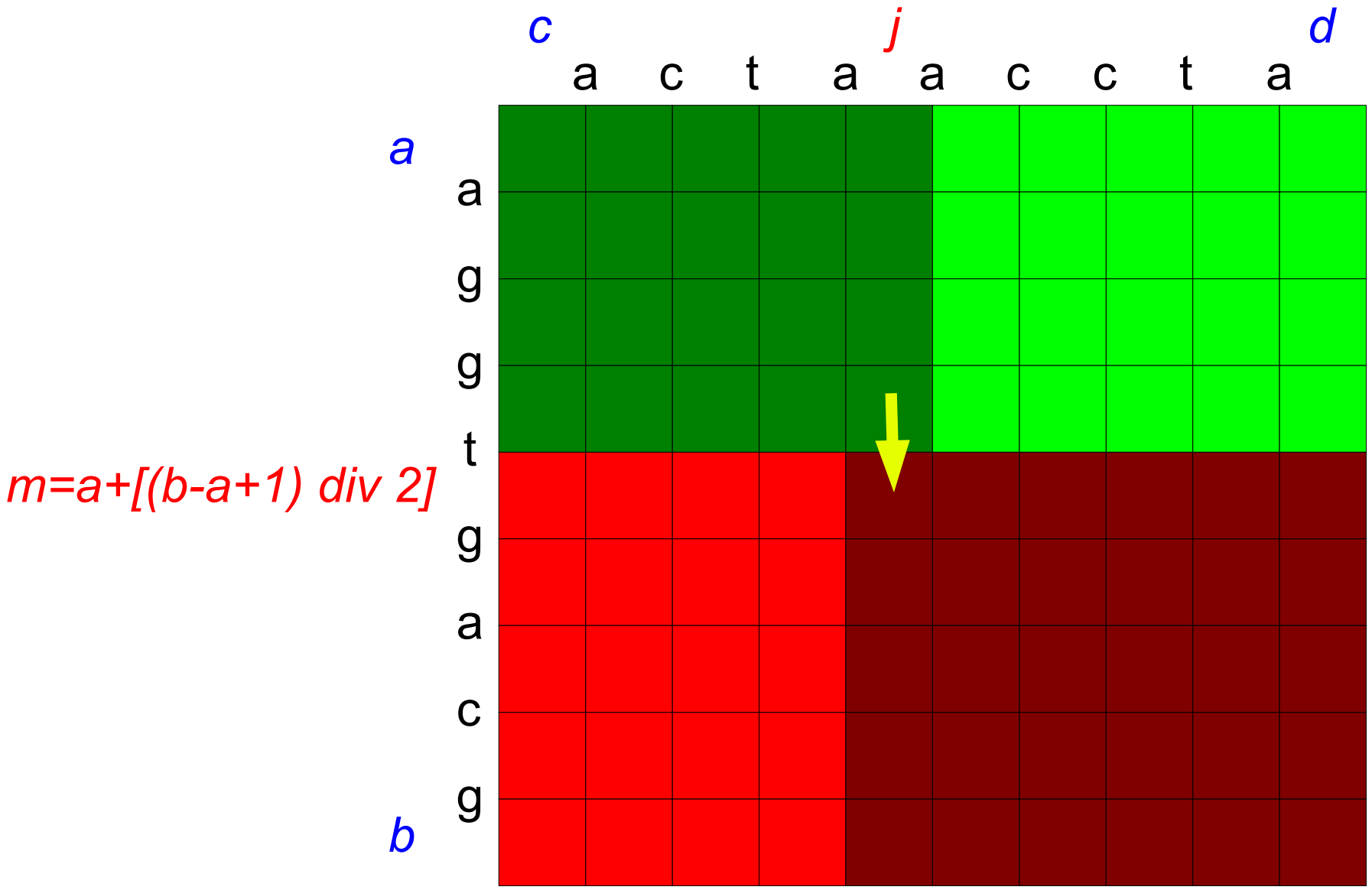
find the optimal alignment of A[a+1..m-1] and B[c+1..j]

find the optimal alignment of A[m+1..b] and B[j+1..d]

find_edge(a,b,c,d,m)=(j,DIAGONAL)



find_edge(a,b,c,d,m)=(j,VERTICAL)



Putting it all together

Overall idea: To find an optimal alignment of A[a+1..b] and B[c+1..d], we call *find_edge(a, b, c, d, a+[(b-a+1) div 2])*, which yields the "middle column", and continue recursively cf. the previous slides by dividing into two subproblems. The base case is when a=b, i.e when we ask for an optimal alignment of the "empty string" and B[c+1..d]. This alignment can be returned immediately as a sequence of gap d-c+1 columns. Computing an optimal alignment of A[1..n] and B[1..n] is done by calling *optimal_alignment(0,n,0,m)*.

Pseudo code:

```
func optimal_alignment(a,b,c,d)
  if a==b then
    return alignment of "empty string" and B[c+1..d]
  else
    "middle edge" = find_edge(a,b,c,d,a+ [(b-a+1) div 2])
    return optimal_alignment("subproblem1")+
           "middle edge" +
           optimal_alignment("subproblem2")
```

Caveats

Be careful with indices: Decide if the input strings A and B are indexed from 0 (as A[0..n-1] and B[0..m-1]) or from 1 (as A[1..n] and B[1..m]). Decide also if the rows and columns in the dynamic programming (and trace-back) table are numbered from 0 or 1.

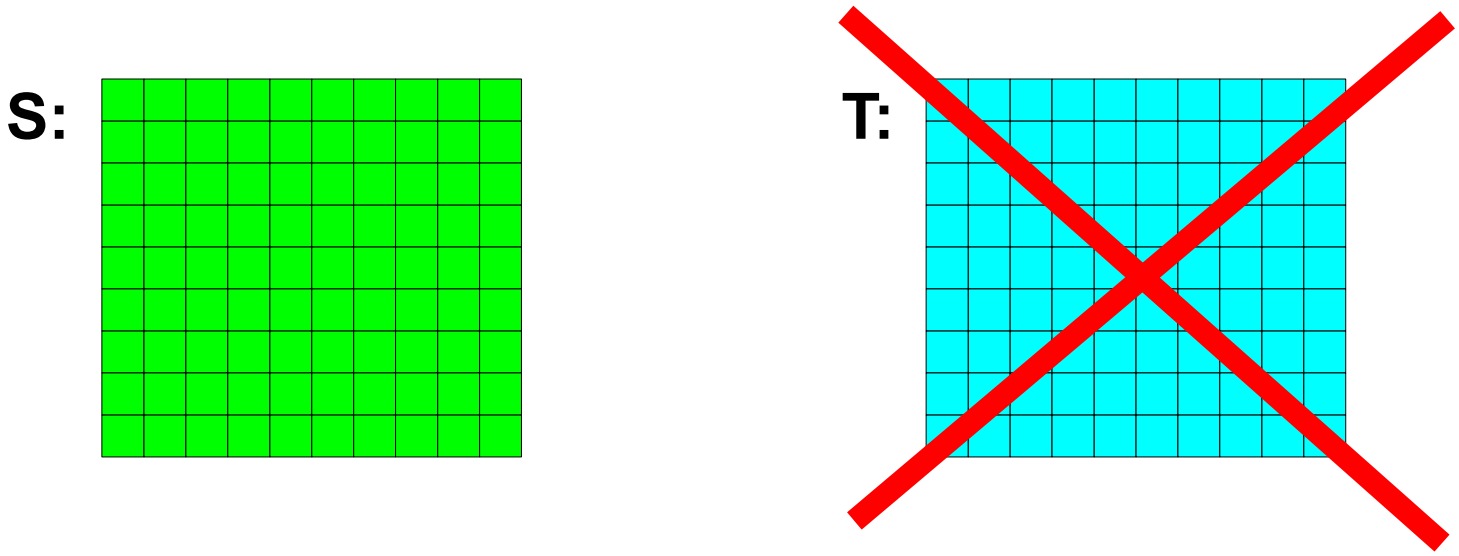
Remember and stick to your decision.

In these slides it is assumed that strings are indexed from 1, and rows/columns are numbered from 0. This means that the sub-table (of the dynamic programming table) from row a to row b (both included) and column c to column d (both included) corresponds to the dynamic programming table for an alignment of A[a+1..b] and B[c+1..d].

Be careful with the base case: Be sure of the definition of your base case (i.e. when a==b as in the pseudo code on the previous slide) and stick with it.

Hirschberg's “find_edge(a,b,c,d,m)”

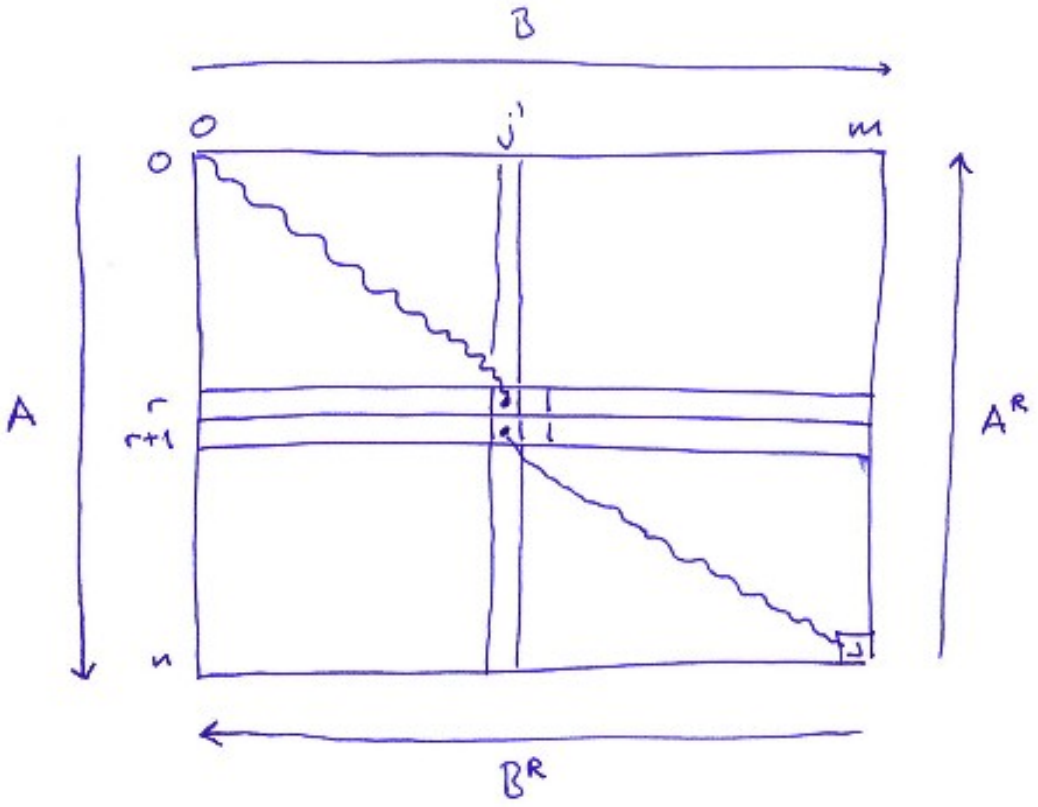
In Hirschberg's original paper, he does not use an addition table to in order to keep track of 'where the optimal path enters the middle row'. Essential he only fills out the dynamic-programming table S.



Observation: Hirschberg utilizes that the optimal cost of an alignment of A and B, OPT(A[1..n], B[1..m]) is equal to the optimal cost of an alignment of the reversed sequences A^R[1..n] = A[n..1] and B^R[1..m] = B[m..1]:

$$\text{OPT}(A[1..n], B[1..m]) = \text{OPT}(A^R[1..n], B^R[1..m])$$

Hirschberg's implementation (1)



Observations

Table S is the dynamic programming table for alignment of *A* and *B*, row *r* can be computed in time O(mr) and space O(m) by filling out the table row-by-row, keeping two rows in memory.

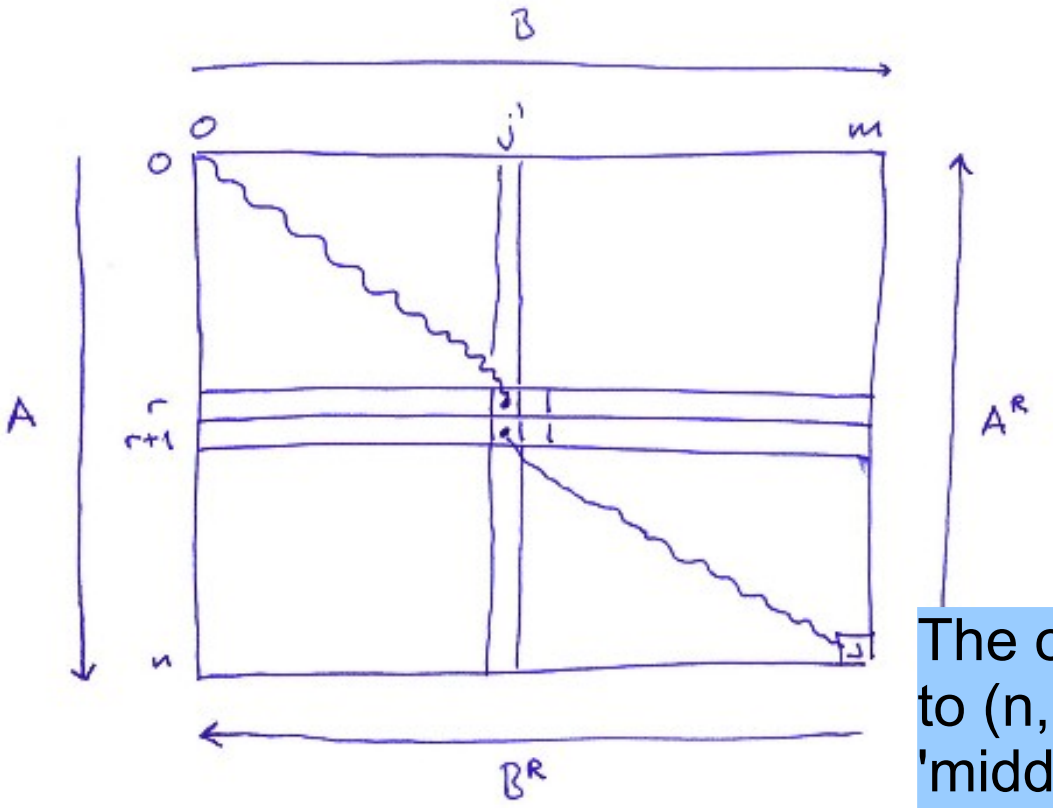
Table S^R is the dynamic programming table for alignment of A^R and B^R, row r+1 of S^R (counting from the top) can be computed in time O(m(n-r)) and space O(m) by filling out the table row-by-row (from the bottom), keeping two rows in memory.

In summary, we can compute 'row r of S' and 'row r+1 of S^R' in time O(nm) and space O(m).

$S[r, j] = OPT(A[1..r], B[1..j])$

$S^R[r+1, j] = OPT(A[r+1..n], B[j..m]) = OPT(A^R[1..n-r], B^R[1..m-j+1])$

Hirschberg's implementation (2)



Observations

Table S is the dynamic programming table for alignment of *A* and *B*, row *r* can be computed in time O(mr) and space O(m) by filling out the table row-by-row, keeping two rows in memory.

Table S^R is the dynamic programming table for alignment of A^R and B^R, row r+1 of S^R (counting from the top) can be computed in time O(m(n-r)) and space O(m) by filling out the table row-by-row (from the bottom), keeping two rows in

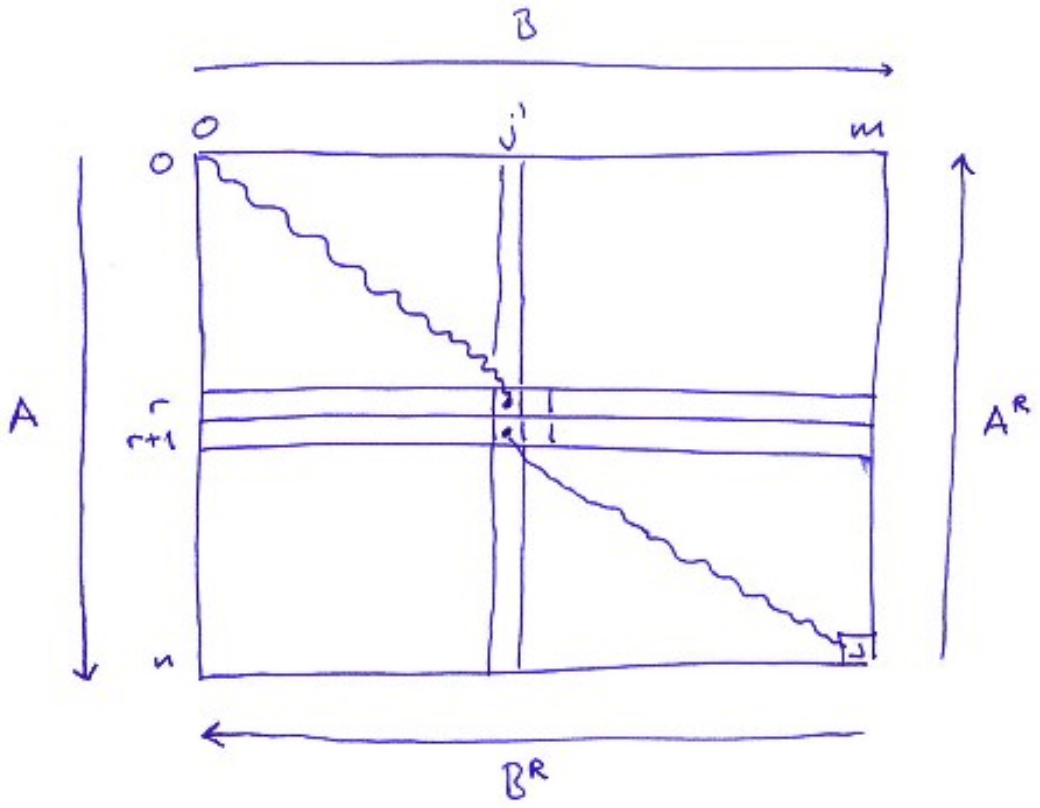
The optimal cost of the path from (0,0) to (n,m) having (j', VERTICAL) as its 'middle edge' is:

$$S[r,j'] + g + S^R[r+1,j']$$

$S[r, j] = OPT(A[1..r], B[1..j])$

$S^R[r+1, j] = OPT(A[r+1..n], B[j..m]) = OPT(A^R[1..n-r], B^R[1..m-j+1])$

Hirschberg's implementation (3)



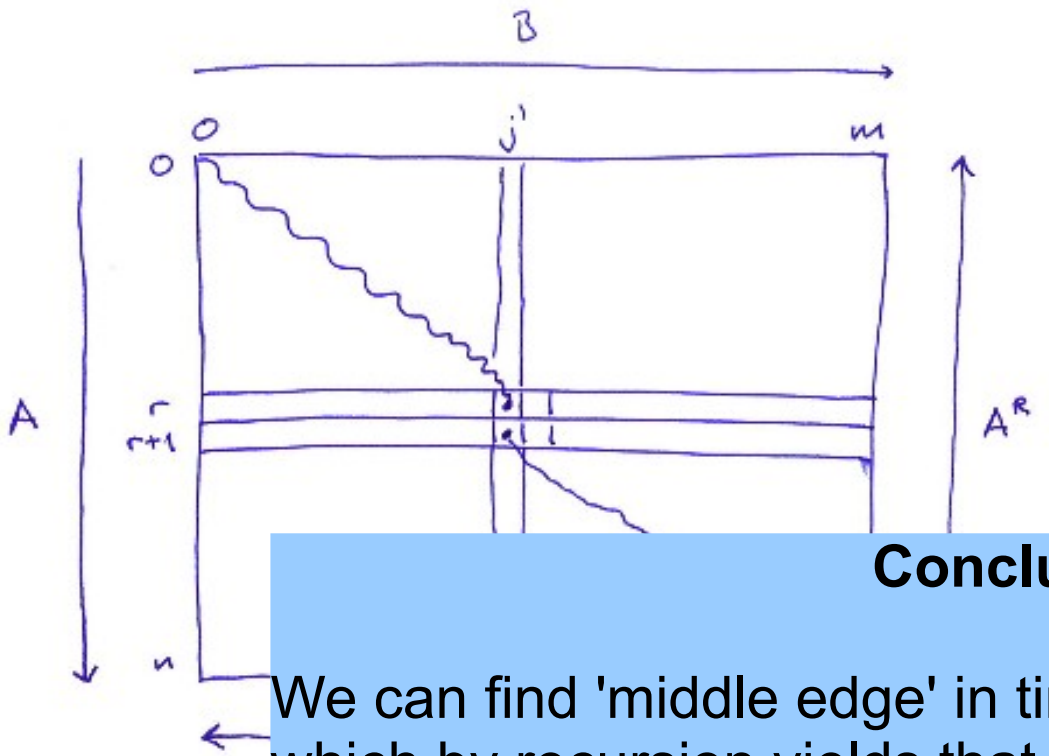

```
// Pseudo code for finding "middle edge" going
// from row r to r+1 of an optimal alignment:

max = 0
middel_edge = (0, VERTICAL)
// Maximize over all (j, VERTICAL)
for j=0 to m do
        curr = S[r,j] + g + S^R[r+1,j]
        if curr > m then
                max = curr
                middle_edge = (j, VERTICAL)
        endif
endfor
// Maximize over all (j, DIAGONAL)
for j=0 to m-1 do
        curr = S[r+j] + s(A[r+1],B[j+1]) + S^R[r+1,j+1]
        if curr > max then
                max = curr
                middle_edge = (j, DIAGONAL)
        endif
endfor
return middle_edge
```

$S[r, j] = OPT(A[1..r], B[1..j])$

$S^R[r+1, j] = OPT(A[r+1..n], B[j..m]) = OPT(A^R[1..n-r], B^R[1..m-j+1])$

Hirschberg's implementation (4)




```
// Pseudo code for finding "middle edge" going
// from row r to r+1 of an optimal alignment:

max = 0
middel_edge = (0, VERTICAL)
// Maximize over all (j, VERTICAL)
for j=0 to m do
        curr = S[r,j] + g + S^R[r+1,j]
        if curr > m then
                max = curr
                middle_edge = (j, VERTICAL)
        endif
                                        S^R[r+1,j+1]
                                        AL)
endfor
return middle_edge
```

Conclusion

We can find 'middle edge' in time O(nm) and space O(m), which by recursion yields that we can compute an optimal alignment in time O(nm) and space O(m)

$S[r, j] = OPT(A[1..r], B[1..j])$

$S^R[r+1, j] = OPT(A[r+1..n], B[j..m]) = OPT(A^R[1..n-r], B^R[1..m-j+1])$