Pairwise sequence comparison

Global pairwise alignment with linear gap cost

Using sequence data

Evolution: DNA evolve by mutations ...

The most frequent mutations are:

Insertion, deletion and substitution of "symbols"

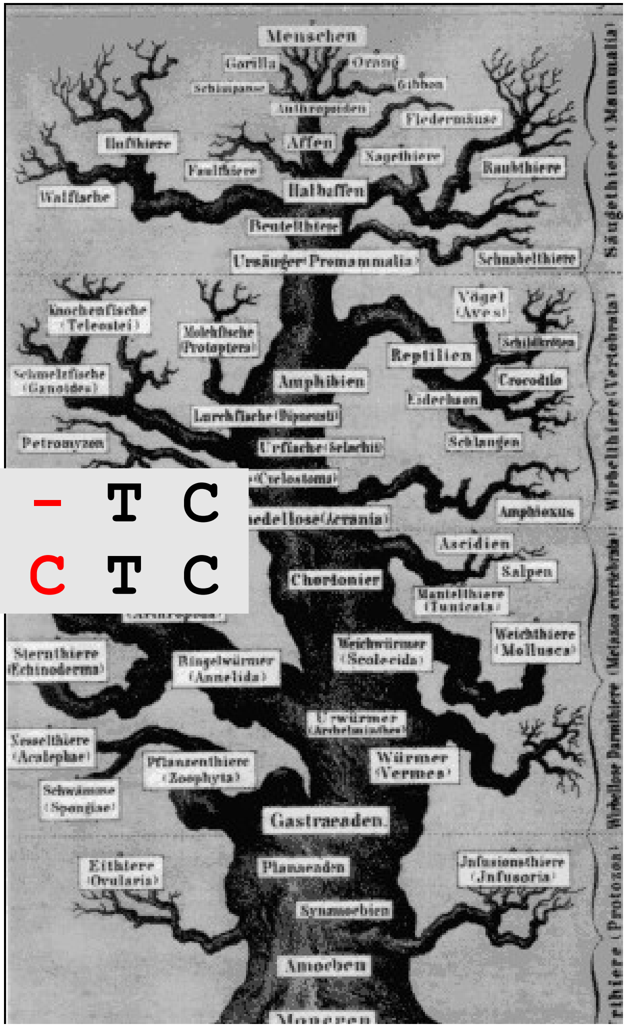
GTTATC $\rightarrow_{\text{ins}}$

GTTACTC $\rightarrow_{\text{del}}$

TT**A**CTC $\rightarrow_{\text{sub}}$

TT**G**CTC

G	T	T	A	-	T	C
-	T	T	G	C	T	C



Parsimony principle: The simplest explanation is a good estimate of evolution, i.e sequences that look similar are related ...

A reasonable question: How similar are two sequences?

Using sequence data

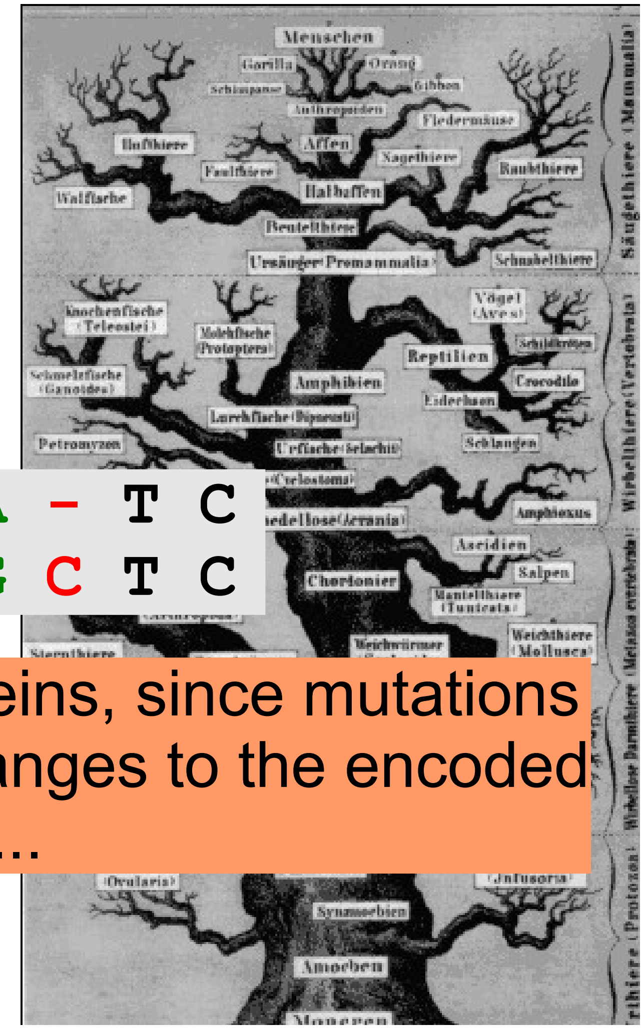
Evolution: DNA evolve by mutations ...

The most frequent mutations are:

Insertion, deletion and substitution of “symbols”

GTTATC $\rightarrow_{ins}$
GTTACTC $\rightarrow_{del}$
TT**A**CTC $\rightarrow_{sub}$
TT**G**CTC

G	T	T	A	-	T	C
-	T	T	G	C	T	C



Note: also applies to proteins, since mutations in coding DNA implies changes to the encoded sequence of amino acids ...

Parsimony principle: T

is a good estimate of evolution, i.e sequences that look similar are related ...

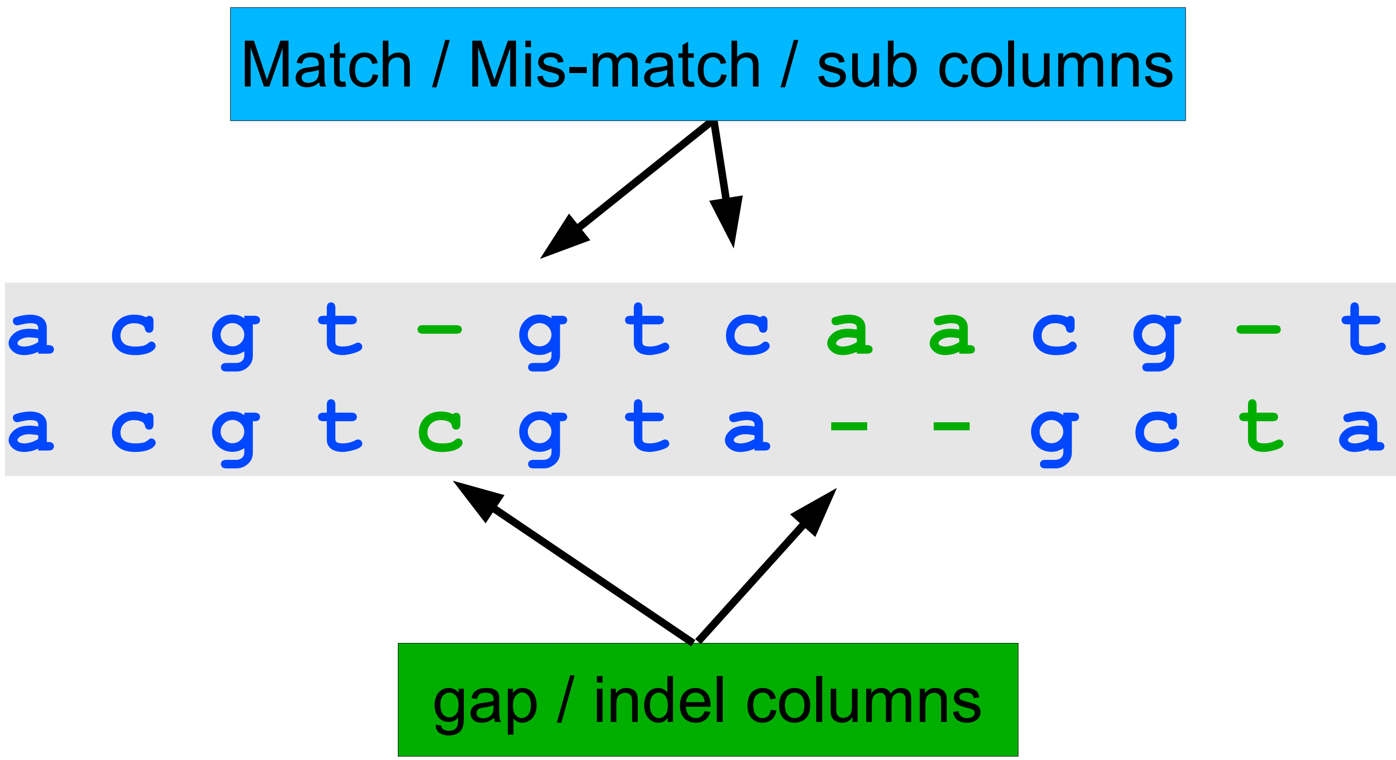
A reasonable question: How similar are two sequences?

What is a pairwise alignment

```
a c g t - g t c a a c g - t
a c g t c g t a - - g c t a
```

A pairwise alignment of acgtgtcaacgt and acgtcgtagcta

What is a pairwise alignment



A pairwise alignment of acgtgtcaacgt and acgtcgtagcta

What is a pairwise alignment

Match / Mis-match / sub columns

```
a c g t - g t c a a c g - t
a c g t c g t a - - g c t a
```

gap / indel columns

An alignment can be interpreted as:

Emphasizing sequence similarity: objective is to maximize the number of match-columns of similar/identical symbols ...

Explaining sequence difference: objective is to minimize the number of indels and subs (of different symbols) ...

What is a pairwise alignment

Match / Mis-match / sub columns

```
a c g t - g t c a a c g - t
a c g t c g t a - - g c t a
```

gap / indel columns

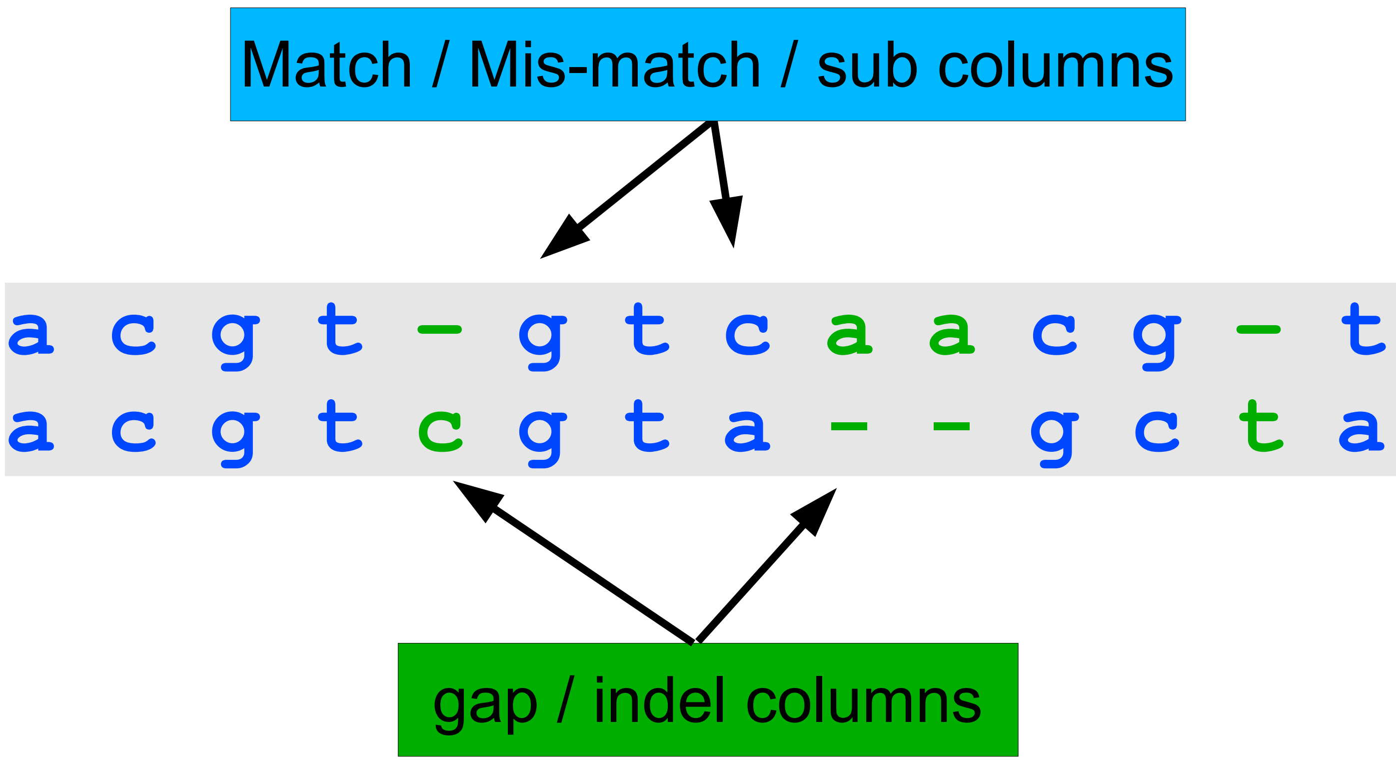
An alignment can be int

$\begin{array}{c} - \\ - \end{array}$ columns are not allowed

Emphasizing sequence similarity: objective is to maximize the number of match-columns of similar/identical symbols ...

Explaining sequence difference: objective is to minimize the number of indels and subs (of different symbols) ...

How to find an optimal alignment



To computationally find an optimal alignment, we must:

Define the cost of an alignment (typically a score matrix and a gap cost)

Define an optimal alignment (typically an alignment of max (or min) cost)

Construct an efficient algorithm for computing an optimal alignment

Cost of an alignment

Score matrix:

	a	c	g	t
a	1	0	0	0
c	0	1	0	0
g	0	0	1	0
t	0	0	0	1

Match / Mis-match / sub columns

a	c	g	t	-	g	t	c	a	a	c	g	-	t
a	c	g	t	c	g	t	a	-	-	g	c	t	a

gap / indel columns

gapcost: 0

Cost of alignment = “sum of the cost of each column”

Cost of an alignment

Score matrix:

	a	c	g	t
a	1	0	0	0
c	0	1	0	0
g	0	0	1	0
t	0	0	0	1

Match / Mis-match / sub columns

```
a c g t - g t c a a c g - t
a c g t c g t a - - g c t a
```

gap / indel columns

gapcost: 0

Cost of alignment = "sum of the cost of each column"

What is the cost of the above alignment?

Cost of an alignment

Match / Mis-match / sub columns

Score matrix:

	a	c	g	t
a	1	0	0	0
c	0	1	0	0
g	0	0	1	0
t	0	0	0	1

```
a c g t - g t c a a c g - t
a c g t c g t a - - g c t a
```

gap / indel columns

gapcost: 0

Cost of alignment = “sum of the cost of each column”

What is the cost of the above alignment? 6

Cost of an alignment

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

Sco
a
c
g 0 0 1 0
t 0 0 0 1

Match / Mis-match / sub columns

```
a c g t - g t c a a c g - t
a c g t c g t a - - g c t a
```

gap / indel columns

gapcost: -1

gapcost: 0

Cost of alignment = "sum of the cost of each column"

What is *now* the cost of the above alignment?

Cost of an alignment

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

Sco…
g 0 0 1 0
t 0 0 0 1

Match / Mis-match / sub columns

```
a c g t - g t c a a c g - t
a c g t c g t a - - g c t a
```

gap / indel columns

gapcost: -1

gapcost: 0

Cost of alignment = "sum of the cost of each column"

What is *now* the cost of the above alignment? 8

Cost of an alignment

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

Sco

g	0	0	1	0
t	0	0	0	1

Match / Mis-match / sub columns

Gap of length 1

```
a c g t - g t c a a c g - t
a c g t c g t a - - g c t a
```

Gap of length 2

gapcost: -1

gapcost: 0

Note about gap cost

In general: cost of “gap block” = g(*k*), where *k* is the gap length

Our examples: g(*k*) = 0·*k* “zero gap cost”

g(*k*) = b·*k* “linear gap cost”

Many programs: g(*k*) = a + b·*k* “affine gap cost”

Cost of an alignment

Match / Mis-match / sub columns

Gap of length 1

```
a c g t - g t c a a c g - t
a c g t c g t a - - g c t a
```

Gap of length 2

```
Score matrix:
   a c g t
a  2 0 0 0
c  0 2 0 0
g  0 0 2 0
t  0 0 0 2
```

```
t  0 0 0 1
```

```
gapcost: -1
```

```
gapcost: 0
```

Note about gap cost

In general:	cost of "gap block" = g(*k*), where *k* is the gap length	
Our examples:	g(*k*) = 0·*k*	"zero gap cost"
	g(*k*) = b·*k*	"linear gap cost"
Many programs:	g(*k*) = a + b·*k*	"affine gap cost"

Computing an optimal alignment

Objective:

Given two sequences *A* and *B*, a score matrix and a gap cost, find an alignment of *A* and *B* of maximum (or minimum) cost.

Computing an optimal alignment

Objective:

Given two sequences *A* and *B*, a score matrix and a gap cost, find an alignment of *A* and *B* of maximum (or minimum) cost.

Example:

Find an optimal alignment of *A*=acgtgtcaacgt and *B*=acgtcgtagcta using:

Score matrix:

	a	c	g	t
a	1	0	0	0
c	0	1	0	0
g	0	0	1	0
t	0	0	0	1

gapcost: 0

Computing an optimal alignment

Objective:

Given two sequences *A* and *B*, a score matrix and a gap cost, find an alignment of *A* and *B* of maximum (or minimum) cost.

Example:

Find an optimal alignment of *A*=acgtgtcaacgt and *B*=acgtcgtagcta using:

Score matrix:

	a	c	g	t
a	1	0	0	0
c	0	1	0	0
g	0	0	1	0
t	0	0	0	1

gapcost: 0

Is this alignment optimal?

```
a c g t - g t c a a c g - t
a c g t c g t a - - g c t a
```

Cost: 6

Computing an optimal alignment

Objective:

Given two sequences *A* and *B*, a score matrix and a gap cost, find an alignment of *A* and *B* of maximum (or minimum) cost.

Example:

Find an optimal alignment of *A*=acgtgtcaacgt and *B*=acgtcgtagcta using:

Score matrix:

	a	c	g	t
a	1	0	0	0
c	0	1	0	0
g	0	0	1	0
t	0	0	0	1

gapcost: 0

Is this alignment optimal?

```
a c g t - g t c a a c g - t
a c g t c g t c - - g c t a
```

Cost: 6

```
a c g t - g t c a a c g t -
a c g t c g t - a g c - t a
```

Cost: 9

Computing an optimal alignment

Objective:

Given two sequences *A* and *B*, a score matrix and a gap cost, find an alignment of *A* and *B* of maximum (or minimum) cost.

Example:

Find an optimal alignment of *A*=acgtgtcaacgt and *B*=acgtcgtagcta using:

```
Score matrix:
   a c g t
 a 1 0 0 0
 c 0 1 0 0
 g 0 0 1 0
 t 0 0 0 1
```

```
gapcost: 0
```

Is this alignment optimal?

```
a c g t - g t c a a c g - t
a c g t c g t a - - g c t a
```

Cost: 6

```
a c g t - g t c a a c g t -
a c g t c g t - a g c - t a
```

Cost: 9

Idea: Compute the cost of every align- ment of *A* and *B* and pick the max ...

Computing an optimal alignment

Objective:

Given two sequences *A* and *B*, a score matrix and a gap cost, find an alignment of *A* and *B* of maximum (or minimum) cost.

Example:

Find an optimal alignment of *A*=acgtgtcaacgt and *B*=acgtcgtagcta using:

Score matrix:

	a	c	g	t
a	1	0	0	0
c	0	1	0	0
g	0	0	1	0
t	0	0	0	1

gapcost: 0

Is this alignment optimal?

```
a c g t - g t c a a c g - t
a c g t c g t c - - g c t a
```

Cost: 6

```
a c g t - g t c a a c g t -
a c g t c g t - a g c - t a
```

Cost: 9

Idea: Compute the cost of every align-ment of *A* and *B* and pick the max ...

How many alignments are there of two strings? Eg. “ag” and “ct”?

Computing an optimal alignment

Objective:

Given two sequences *A* and *B*, a score matrix and a gap cost, find an alignment of *A* and *B* of maximum (or minimum) cost.

Example:

Find an optimal alignment of *A*=acgtgtcaacgt and *B*=acgtcgtagcta using:

Score matrix:

	a	c	g	t
a	1	0	0	0
c	0	1	0	0
g	0	0	1	0
t	0	0	0	1

gapcost: 0

Is this alignment optimal?

~~a c g t - g t c a a c g - t~~
~~a c g t c g t c - - g c t a~~

~~Cost: 6~~

```
a c g t - g t c a a c g t -
a c g t c g t - a g c - t a
```

Cost: 9

~~**Idea:** Compute the cost of every align-ment of *A* and *B* and pick the max ...~~

How many alignments are there of two strings? Eg. “ag” and “ct”?

Computing an optimal alignment

Objective:

Given two sequences *A* and *B*, a score matrix and a gap cost, find an alignment of *A* and *B* of maximum (or minimum) cost.

```
GGCCTAAAGG CGCCGGTCTT TCGTACCCCA AAATCTCGGC ATTTTAAGAT
AAGTGAGTGT TGCGTTACAC TAGCGATCTA CCGCGTCTTA TACTTAAGCG
TATGCCCAGA TCTGACTAAT CGTGCCCCCG GATTAGACGG GCTTGATGGG
AAAGAACAGC TCGTCTGTTT ACGTATAAAC AGAATCGCCT GGGTTCGC

GGGCTAAAGG TTAGGGTCTT TCACACTAAA GAGTGGTGCG TATCGTGGCT
AATGTACCGC TTCTGGTATC GTGGCTTACG GCCAGACCTA CAAGTACTAG
ACCTGAGAAC TAATCTTGTC GAGCCTTCCA TTGAGGGTAA TGGGAGAGAA
CATCGAGTCA GAAGTTATTC TTGTTTACGT AGAATCGCCT GGGTCCGC
```

Needs an efficient computational tool. One of the first and most studied problems in bioinformatics ...

Computing an optimal alignment

V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 1966.

T. K. Vintsyuk. Speech discrimination by dynamic programming. *Kibernetika*, 1968.

S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 1970.

D. Sankoff. Matching sequence under deletion/insertion constraints. *Proceedings of the National Academy of Science of the USA*, 1972.

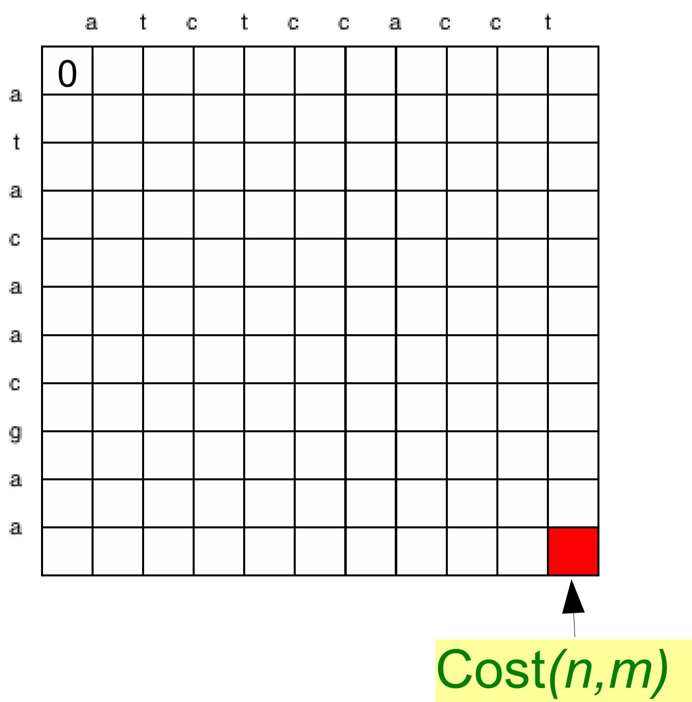
R. A. Wagner and M. J. Fisher. The String to String Correction Problem. *Journal of the ACM*, 1973.

P. H. Sellers. On the theory and computation of evolutionary distance. *SIAM Journal of Applied Mathematics*, 1974.

... and many more ...

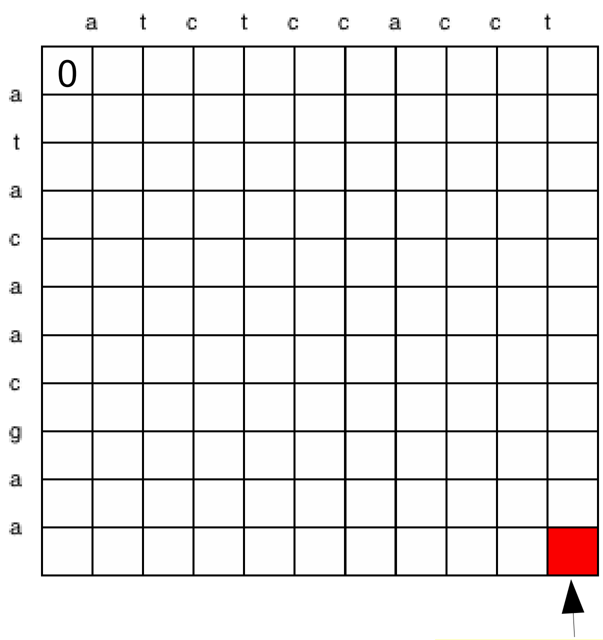
Constructing an algorithm

Problem: What is the cost, Cost(i, j), of an optimal alignment of the first i symbols in A, $A[1..i]$, and the first j symbols in B, $B[1..j]$?



Constructing an algorithm

Problem: What is the cost, Cost*(i, j)*, of an optimal alignment of the first *i* symbols in *A*, *A*[1..*i*], and the first *j* symbols in *B*, *B*[1..*j*]?



Cost*(n,m)*

```
ATACAACGA  A
ATCTCCACC  T
```

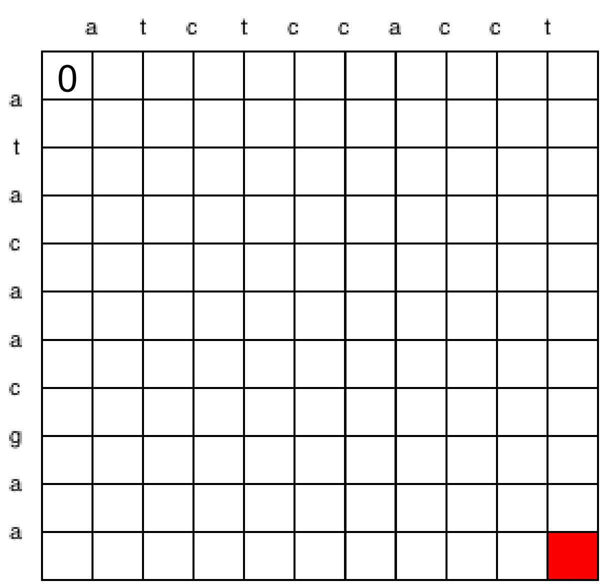
```
ATACAACGA  A
ATCTCCACCT -
```

```
ATACAACGAA -
ATCTCCACC  T
```

Solution: Look at the last column, there are 3 cases, pick the best

Constructing an algorithm

Problem: What is the cost, Cost*(i, j)*, of an optimal alignment of the first *i* symbols in *A*, *A*[1..*i*], and the first *j* symbols in *B*, *B*[1..*j*]?



Cost*(n,m)*

```
ATACAACGA A
ATCTCCACC T
```

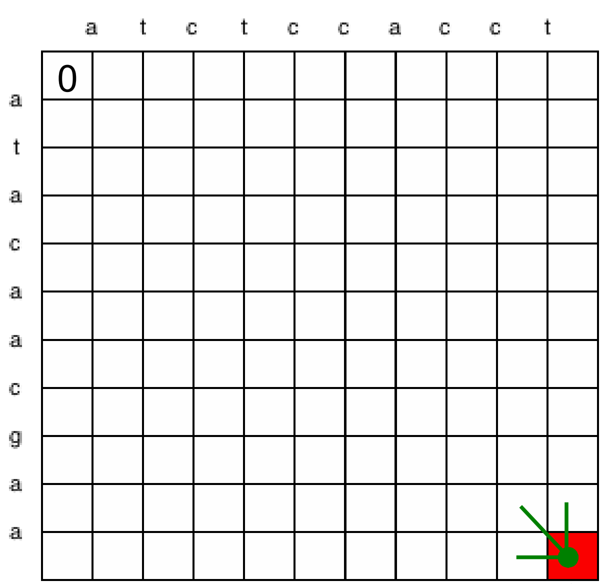
Cost(*n*-1,*m*-1) + subcost(*A*[*n*], *B*[*m*])

An optimal alignment of the first *n*-1 symbols in A and the first *m*-1 symbols in *B*

Solution: Look at the last column, there are 3 cases, pick the best

Constructing an algorithm

Problem: What is the cost, Cost(i, j), of an optimal alignment of the first i symbols in A, $A[1..i]$, and the first j symbols in B, $B[1..j]$?



Cost(n,m)

ATACAACGA A
ATCTCCACC T

Cost(n-1,m-1) + subcost($A[n]$, $B[m]$)

ATACAACGA A
ATCTCCACCA –

Cost(n-1,m) + gapcost

ATACAACGCA –
ATCTCCACC T

Cost(n,m-1) + gapcost

Solution: Look at the last column, there are 3 cases, pick the best

A simple recursive solution

Implementing the algorithm

Possible cases:

ATACAACGA	A
ATCTCCACC	T

ATACAACGA	A
ATCTCCACCT	-

ATACAACGAA	-
ATCTCCACC	T

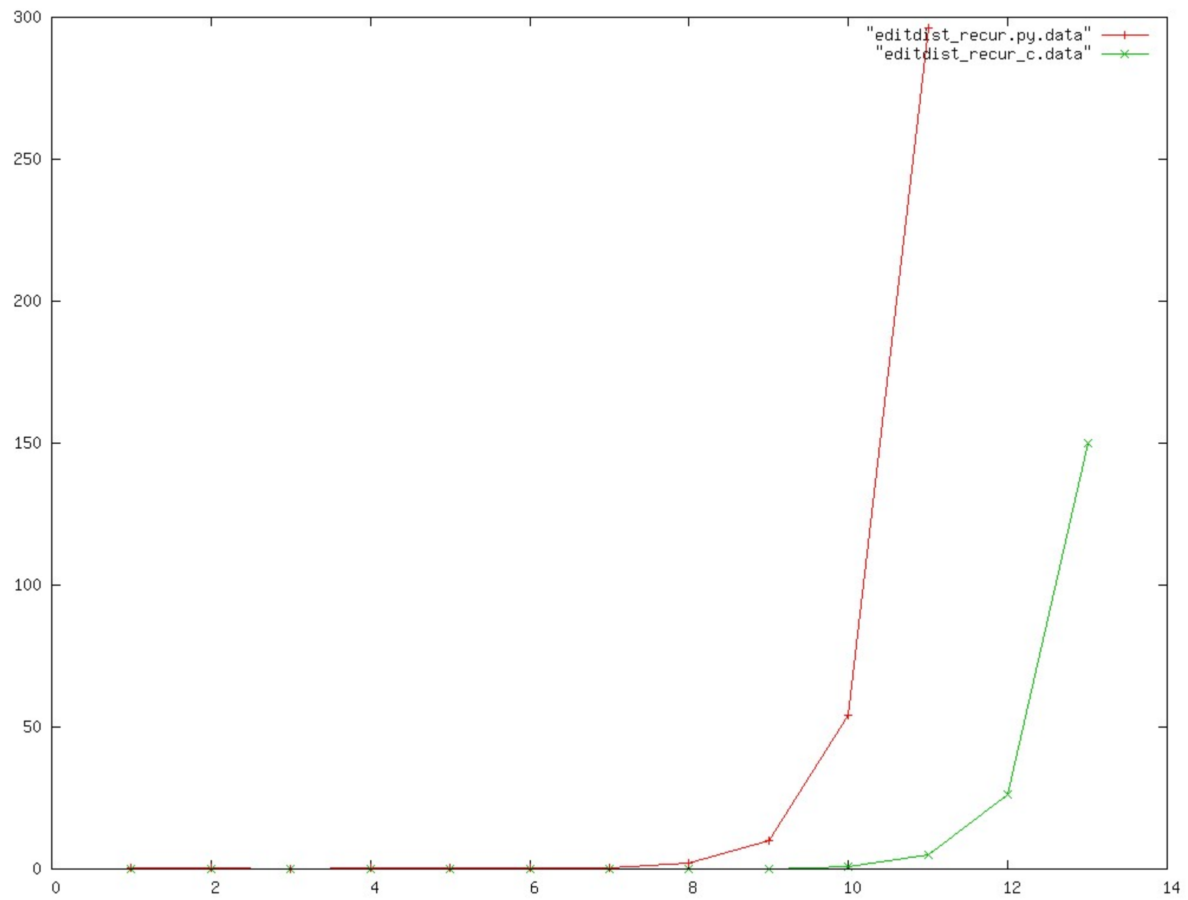
or *nothing* Cost(0,0)=0

```
func Cost(i,j):
    v1 = v2 = v3 = v4 = undef
    if (i > 0) and (j > 0) then
        v1 = Cost(i-1, j-1) + d(A[i],B[j])
    if (i > 0) and (j >= 0) then
        v2 = Cost(i-1, j) + g
    if (i >= 0) and (j > 0) then
        v3 = Cost(i, j-1) + g
    if (i = 0) and (j = 0) then
        v4 = 0
    return max(v1,v2,v3,v4)
end
print Cost(n,m)
```

Is it a good solution? Is it correct? Is it efficient?

Experiment

Measure the running time for comparing sequences of length 1, 2, 3, ...



Slow! What is the bottleneck?

Analysing the algorithm

```
func Cost(i,j):
     v1 = v2 = v3 = v4 = undef
     if (i > 0) and (j > 0) then
          v1 = Cost(i-1, j-1) + d(A[i],B[j])
     if (i > 0) and (j >= 0) then
          v2 = Cost(i-1, j) + g
     if (i >= 0) and (j > 0) then
          v3 = Cost(i, j-1) + g
     if (i = 0) and (j = 0) then
          v4 = 0
     return max(v1,v2,v3,v4)
end
print Cost(n,m)
```

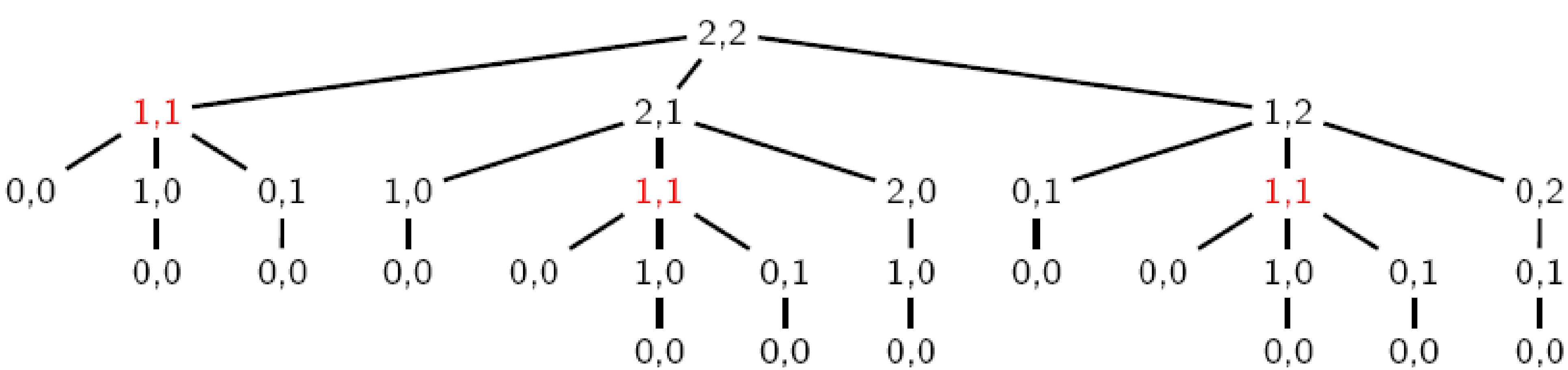
Computing Cost(2,2) involves

Cost(1,1), Cost(1,2), Cost(2,1) ...

We compute the same thing again and again ...

Rule of thumb:

Remember what you have done!!



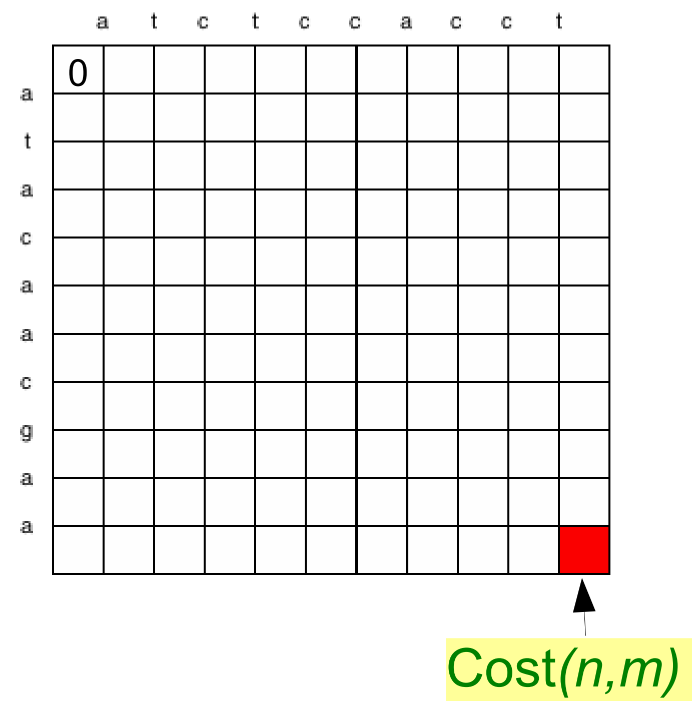
Analysing the algo

```
func Cost(i,j):
     v1 = v2 = v3 = v4 = undef

     if (i > 0) and (j > 0) then
          v1 = Cost(i-1, j-1) + d(A[i],B[j])

     if (i > 0) and (j >= 0) then
          v2 = Cost(i-1, j) + g

     if (i >= 0) and (j > 0) then
          v3 = Cost(i, j-1) + g

     if (i = 0) and (j = 0) then
          v4 = 0

     return max(v1,v2,v3,v4)
end
print Cost(n,m)
```

Computing C

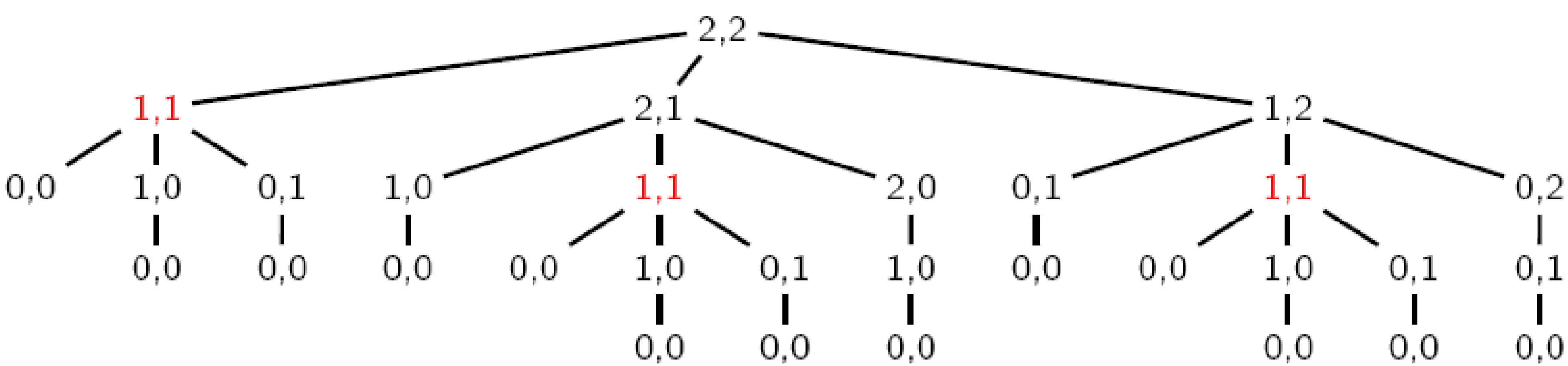
Cost(1,1), Co

We compute
and again ...



Rule of thumb:

Remember what you have done!!



A better algorithm

Store intermediate results – *Dynamic programming*

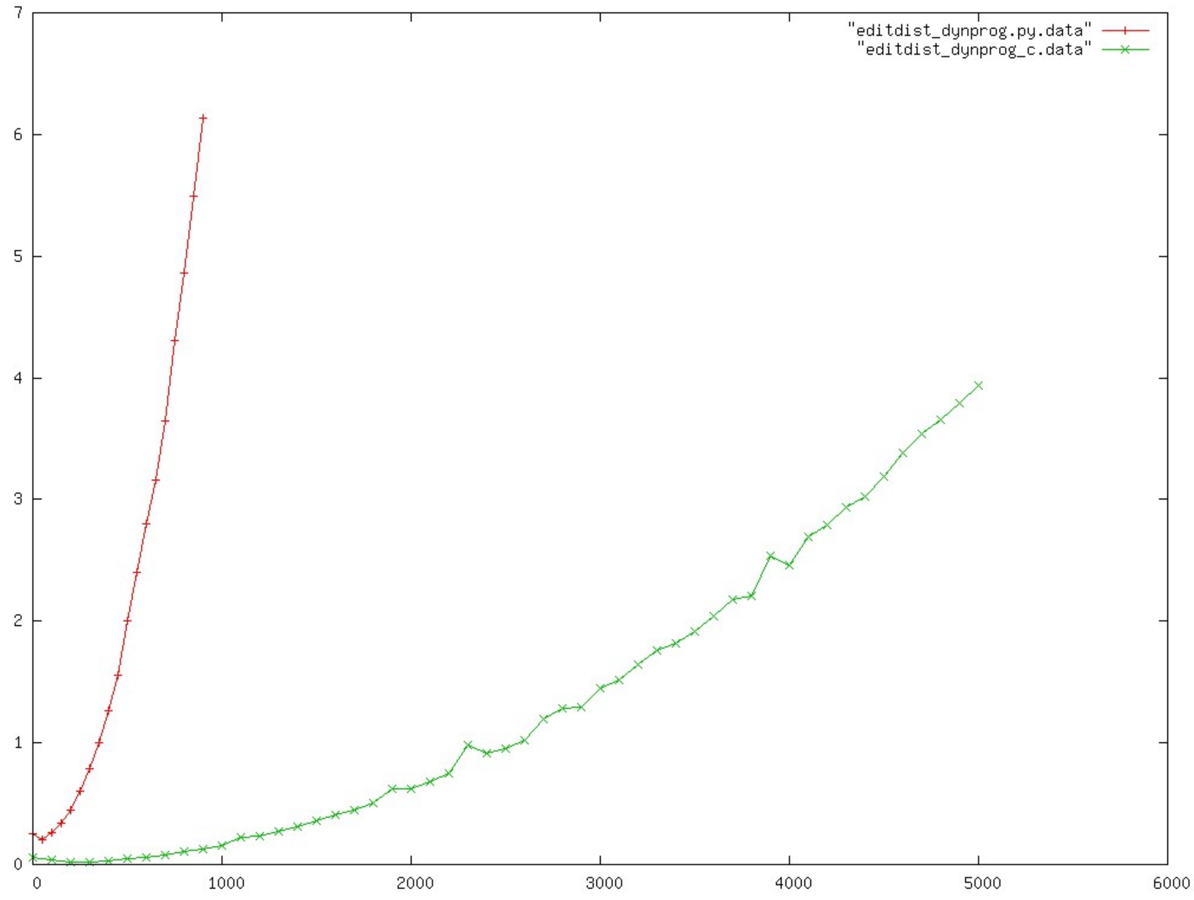
```
func Cost(i,j):

    v1 = v2 = v3 = v4 = undef
    if (i > 0) and (j > 0) then
        v1 = Cost(i-1, j-1) + d(A[i],B[j])
    if (i > 0) and (j >= 0) then
        v2 = Cost(i-1, j) + g
    if (i >= 0) and (j > 0) then
        v3 = Cost(i, j-1) + g
    if (i = 0) and (j = 0) then
        v4 = 0



    return max(v1,v2,v3,v4)
end

print Cost(n,m)
```

```
func Cost(i, j):
    if T[i,j] = undef then
        v1 = v2 = v3 = v4 = undef
        if (i > 0) and (j > 0) then
            v1 = Cost(i-1, j-1) + d(A[i],B[j])
        if (i > 0) and (j >= 0) then
            v2 = Cost(i-1, j) + g
        if (i >= 0) and (j > 0) then
            v3 = Cost(i, j-1) + g
        if (i = 0) and (j = 0) then
            v4 = 0
        T[i,j] = max(v1,v2,v3,v4)
    endif
    return T[i,j]
end
T[0..n][0..m]=undef
print Cost(n,m)
```

How does this influence the running time?

Experiment

Measure the running time for comparing sequences of length 1, 2, 3, ...

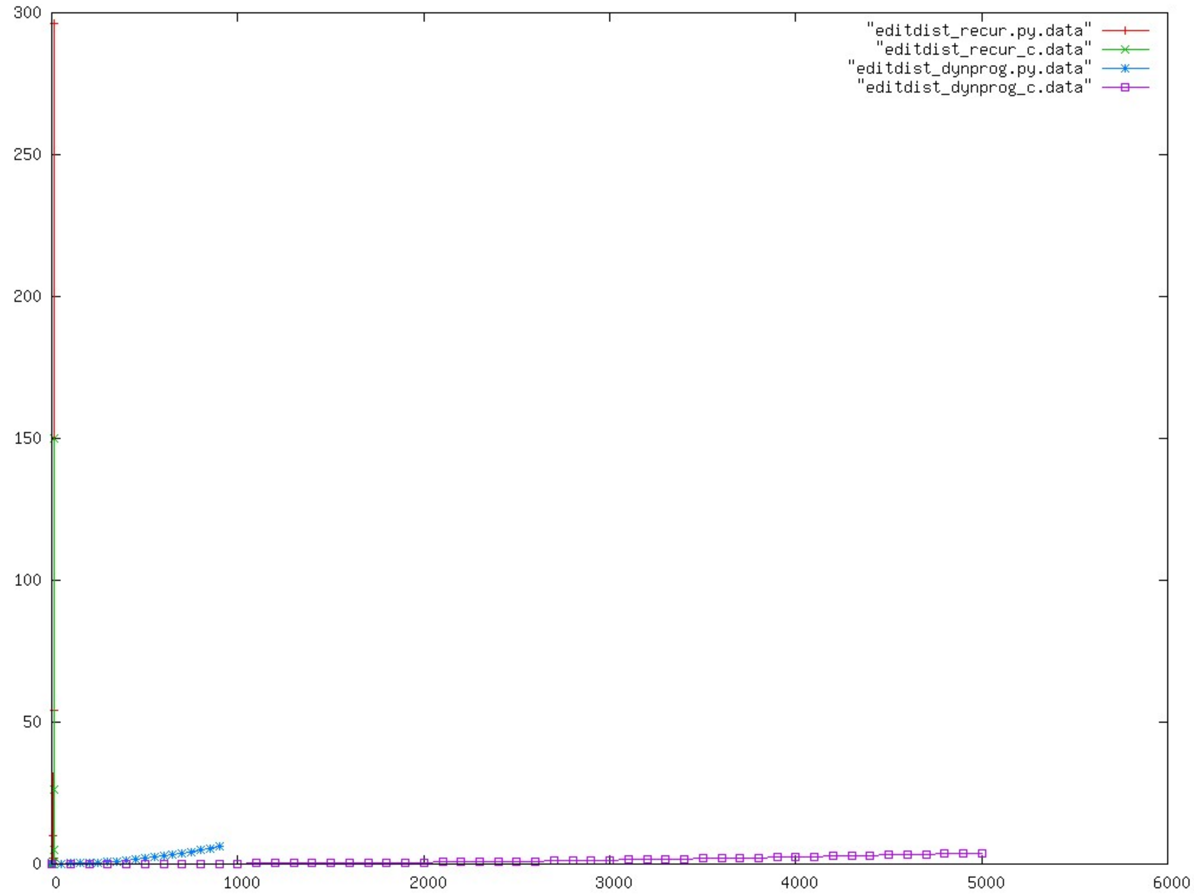


Much faster!!

The running time and space usage is prop. to the size of the table O(*nm*)

Experiment

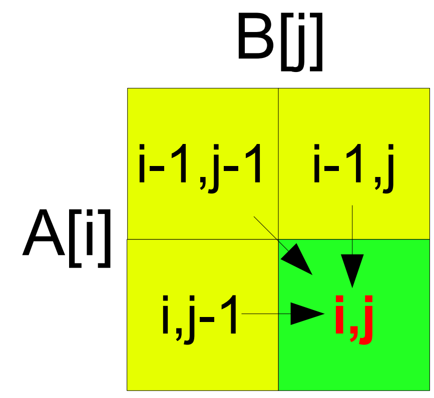
The running time of both algorithms implemented in C and Python



The improved algorithm makes the solution usable in practice

Global alignment recursion

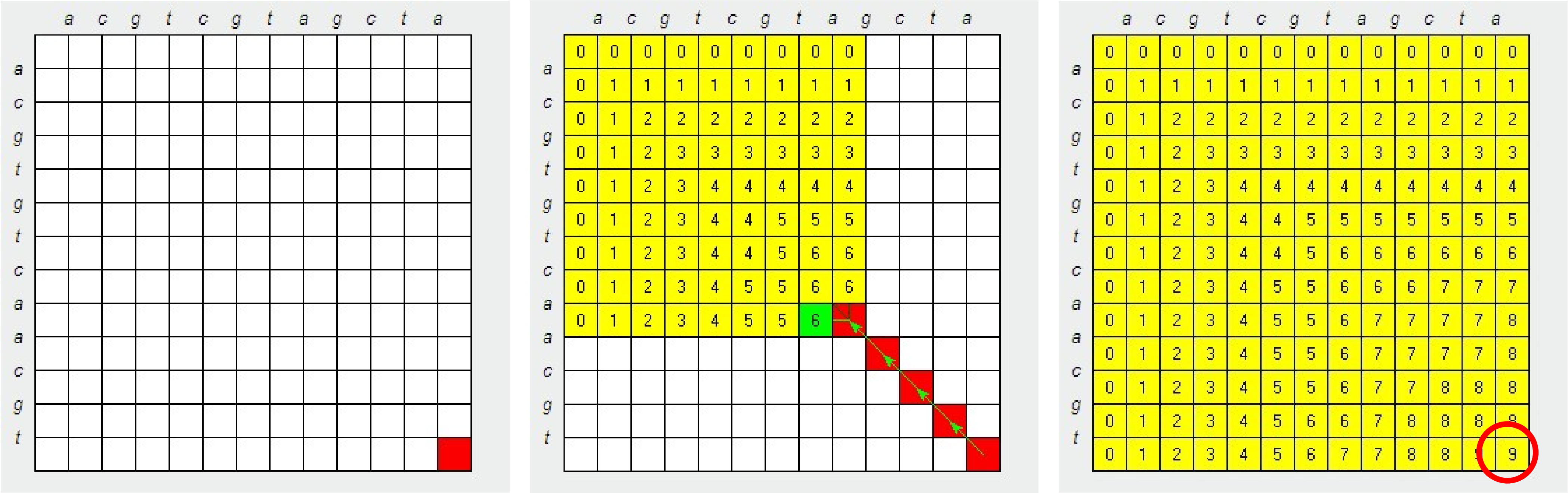
$$\text{Cost}(i, j) = \max \begin{cases} \text{Cost}(i-1, j-1) + \text{subcost}(A[i\,], B[j\,]) \\ \text{Cost}(i-1, j) + \text{gapcost} \\ \text{Cost}(i, j-1) + \text{gapcost} \\ 0 \text{ if } i=0 \text{ and } j=0 \end{cases}$$



To compute the score of an optimal alignment of *A*[1..*n*] and *B*[1..*m*], fill out an (*n*+1) x (*m*+1) table cf. above recursion.

The optimal alignment score is in entry (*n*,*m*).

Basic idea: Filling out a table



The algorithm essentially fills out a table (dynamic programming)

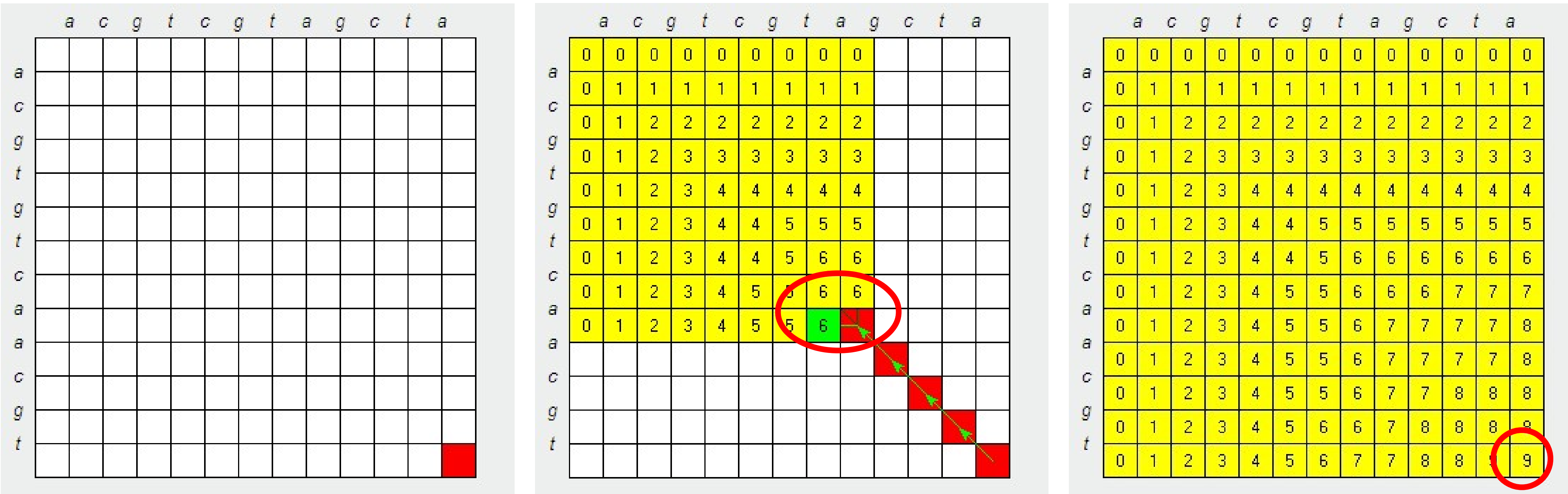
Observation: The value in a cell depends on three of its neighbor cells (left, upper-diagonal, and above) ...

Score matrix:

	a	c	g	t
a	1	0	0	0
c	0	1	0	0
g	0	0	1	0
t	0	0	0	1

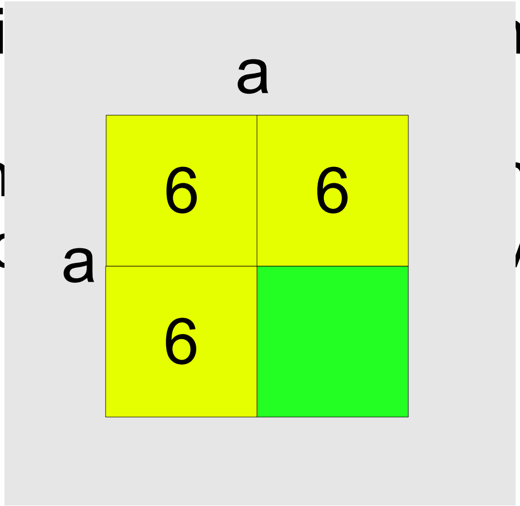
gapcost: 0

Basic idea: Filling out a table



The algorithm essentially fi … amic programming)

Observation: The value in … three of its
neighbor cells (left, upper-… ve) ...

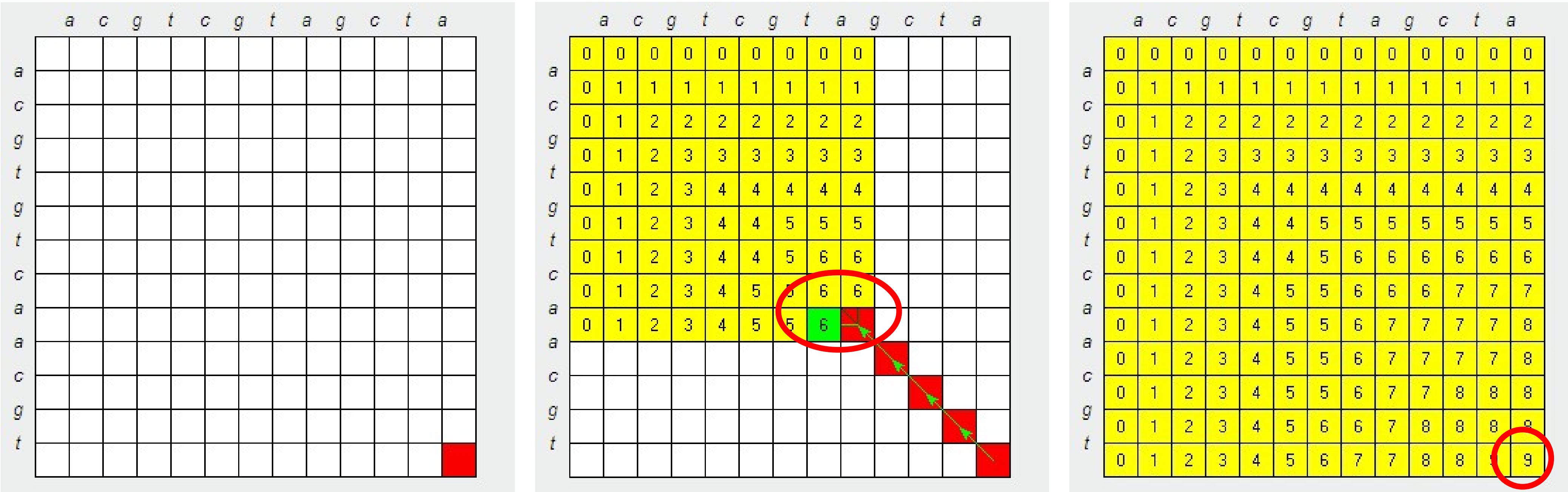


Score matrix:

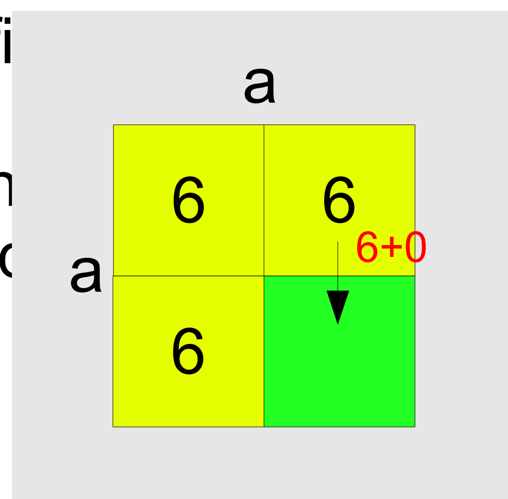
	a	c	g	t
a	1	0	0	0
c	0	1	0	0
g	0	0	1	0
t	0	0	0	1

gapcost: 0

Basic idea: Filling out a table



The algorithm essentially fi amic programming)



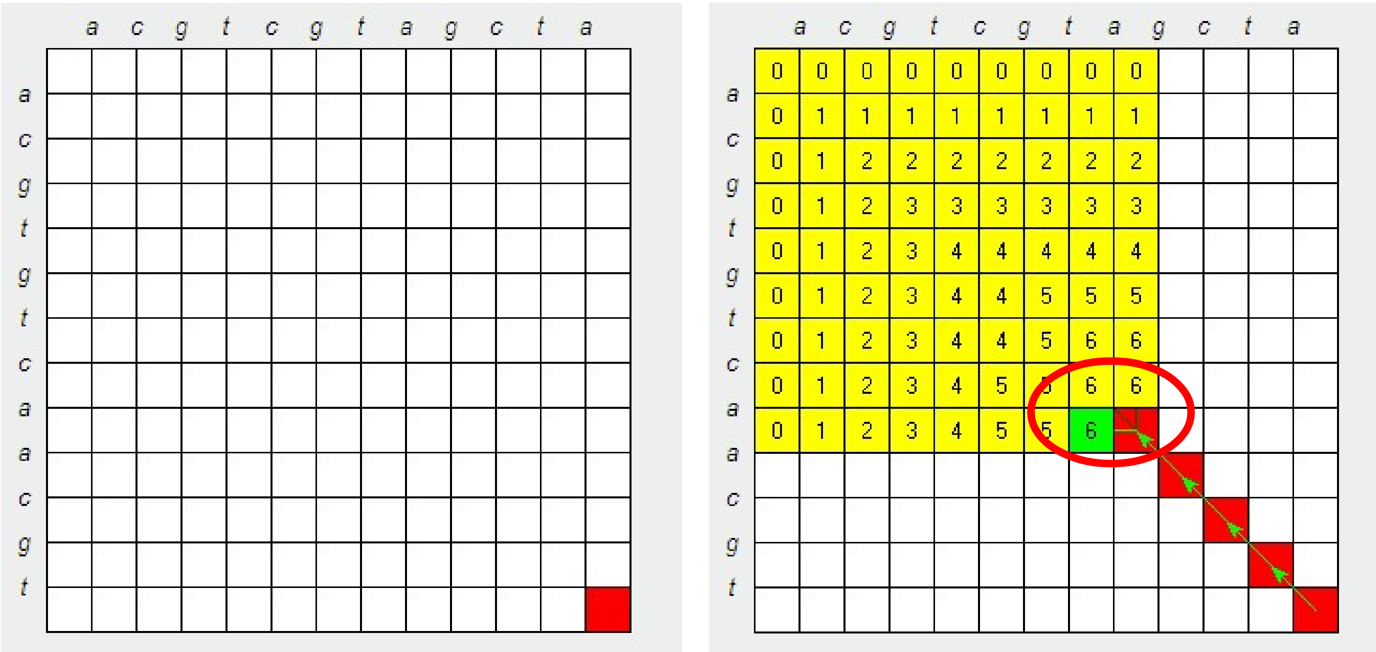
Observation: The value in three of its
neighbor cells (left, upper-c ve) ...

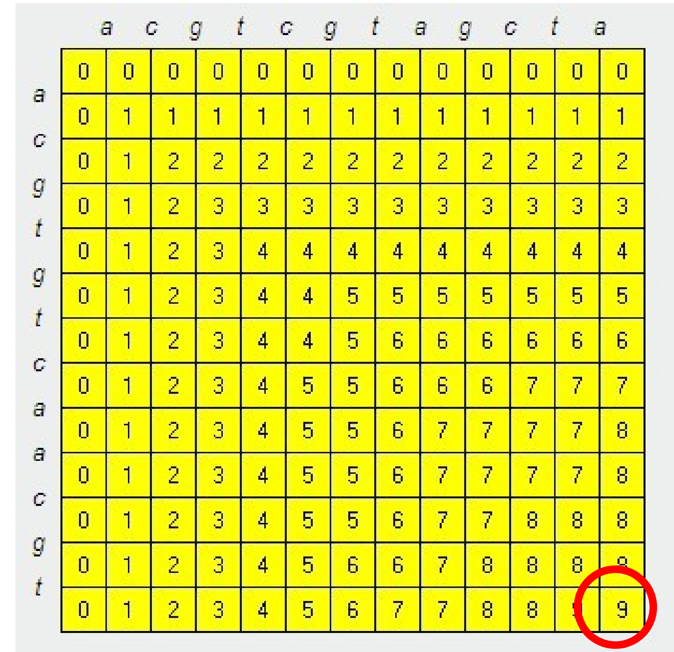
Score matrix:

	a	c	g	t
a	1	0	0	0
c	0	1	0	0
g	0	0	1	0
t	0	0	0	1

gapcost: 0

Basic idea: Filling out a table

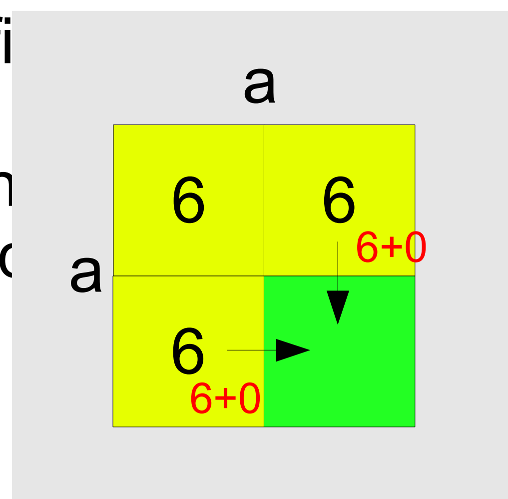




		a	c	g	t	c	g	t	a	g	c	t	a
	0	0	0	0	0	0	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1	1	1	1	1	1	1
c	0	1	2	2	2	2	2	2	2	2	2	2	2
g	0	1	2	3	3	3	3	3	3	3	3	3	3
t	0	1	2	3	4	4	4	4	4	4	4	4	4
g	0	1	2	3	4	4	5	5	5	5	5	5	5
t	0	1	2	3	4	4	5	6	6	6	6	6	6
c	0	1	2	3	4	5	5	6	6	6	7	7	7
a	0	1	2	3	4	5	5	6	7	7	7	7	8
a	0	1	2	3	4	5	5	6	7	7	7	7	8
c	0	1	2	3	4	5	5	6	7	7	8	8	8
g	0	1	2	3	4	5	6	6	7	8	8	8	[illegible]
t	0	1	2	3	4	5	6	7	7	8	8	[illegible]	9

The algorithm essentially fi … namic programming)

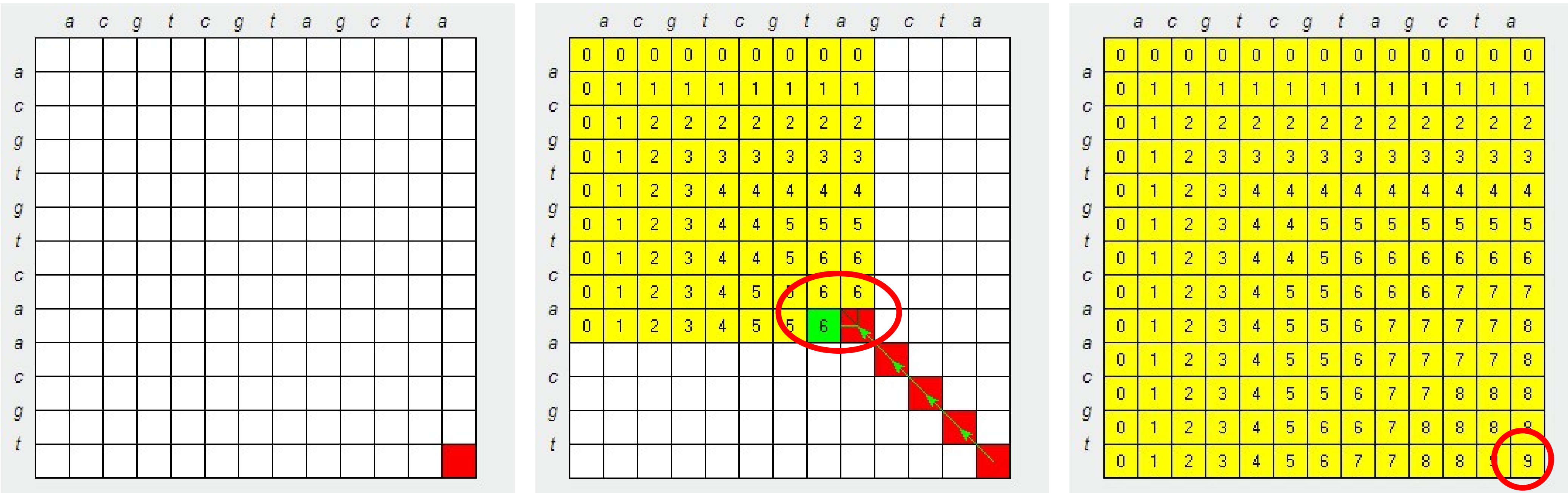
Observation: The value in … three of its neighbor cells (left, upper-… ve) ...



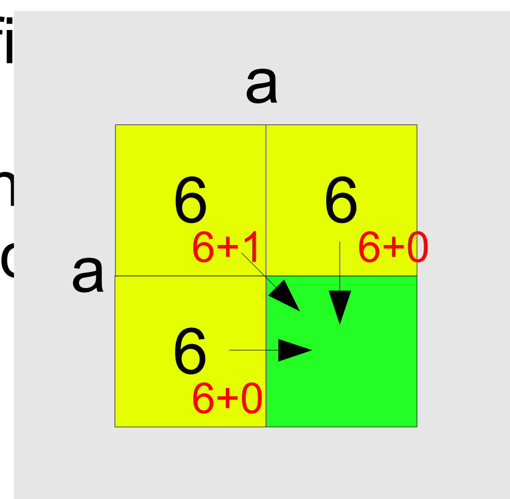
Score matrix:	a	c	g	t
a	1	0	0	0
c	0	1	0	0
g	0	0	1	0
t	0	0	0	1

gapcost: 0

Basic idea: Filling out a table



The algorithm essentially fi … amic programming)



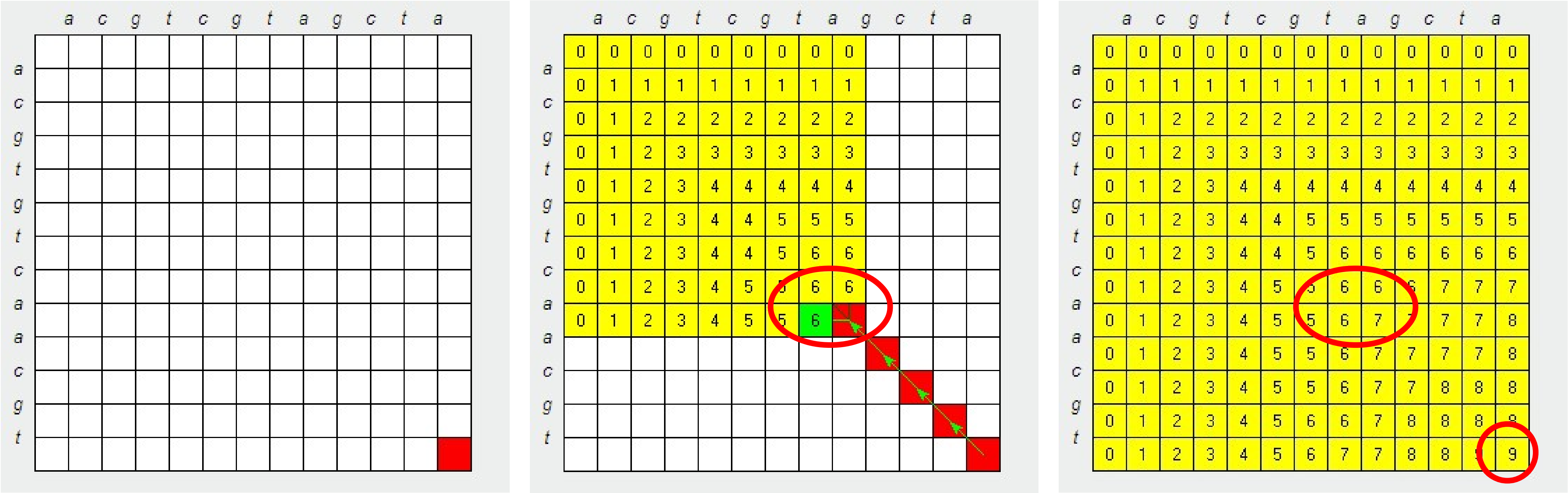
Observation: The value in … three of its
neighbor cells (left, upper-… ve) ...

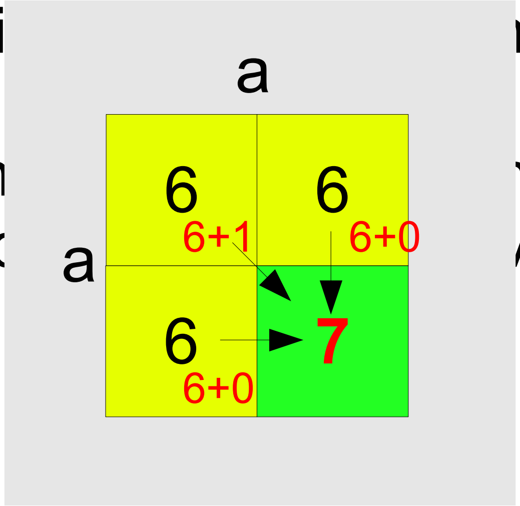
Score matrix:

	a	c	g	t
a	1	0	0	0
c	0	1	0	0
g	0	0	1	0
t	0	0	0	1

gapcost: 0

Basic idea: Filling out a table



The algorithm essentially fi 

namic programming)

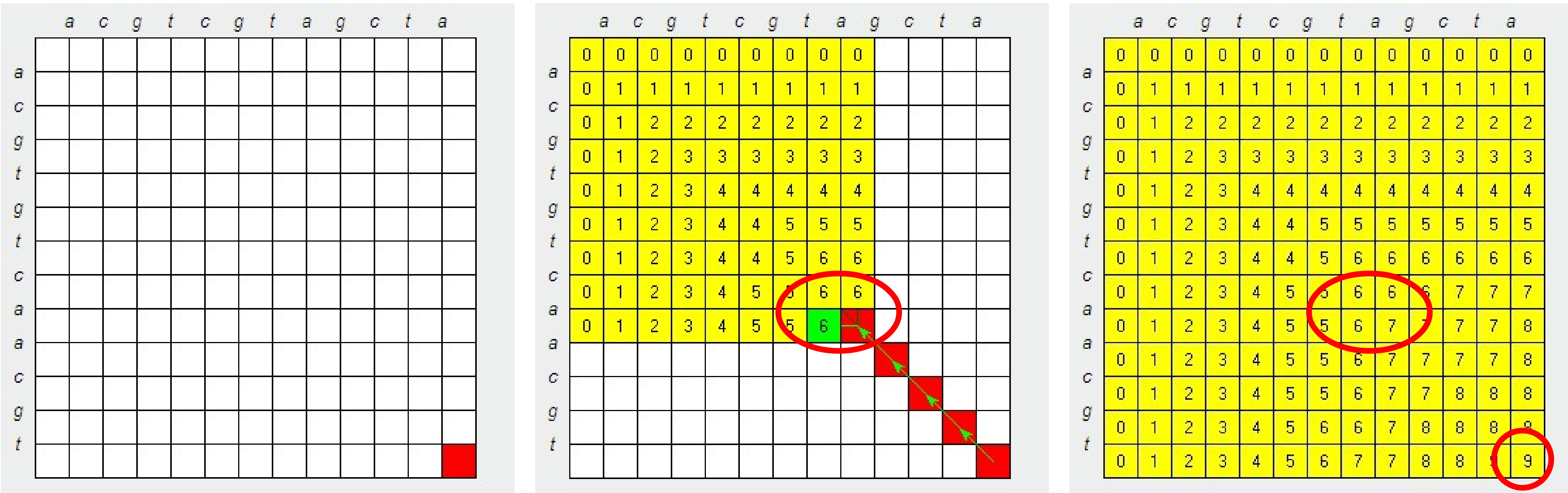
Observation: The value in ... three of its neighbor cells (left, upper-... ve) ...

Score matrix:

	a	c	g	t
a	1	0	0	0
c	0	1	0	0
g	0	0	1	0
t	0	0	0	1

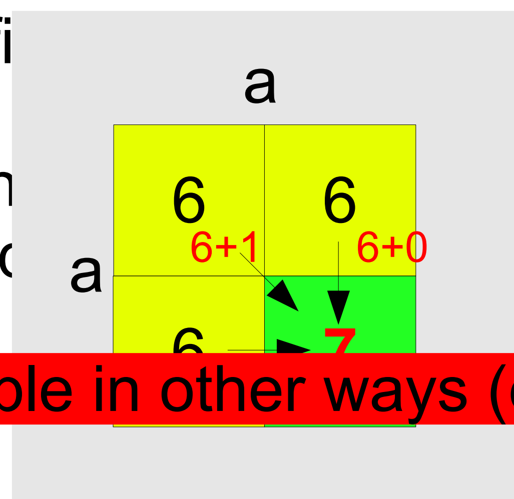
gapcost: 0

Basic idea: Filling out a table



The algorithm essentially fi… …amic programming)

Observation: The value in … three of its neighbor cells (left, upper-… …ve) ...



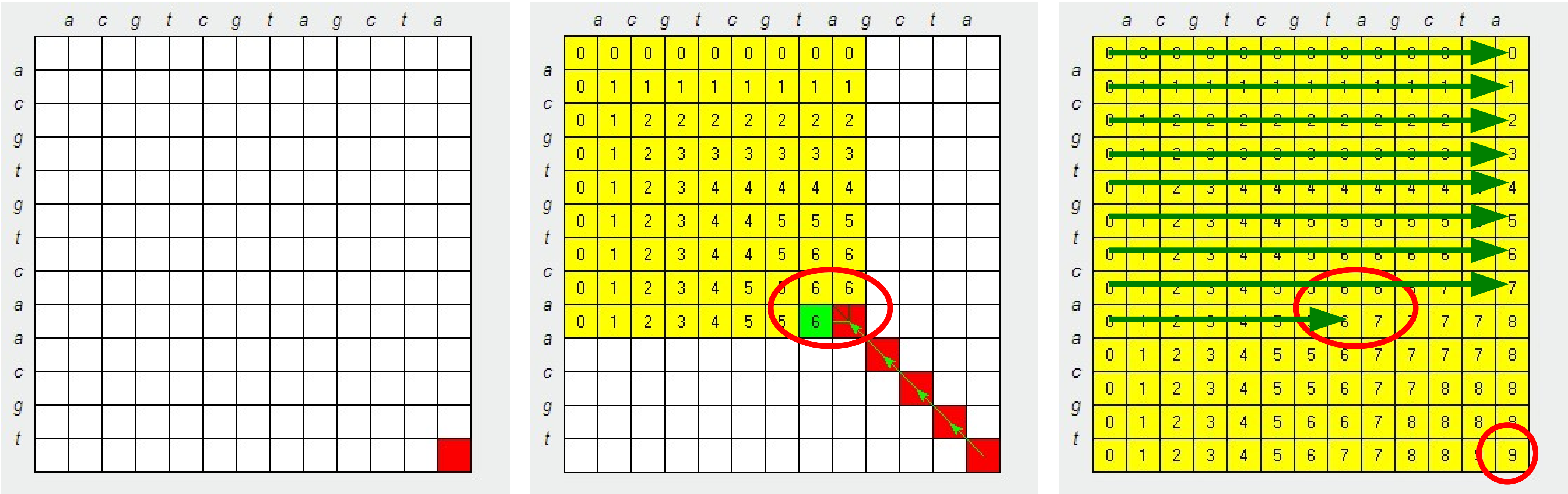
Can we fill out the table in other ways (e.g. non-recursively)?

Score matrix:

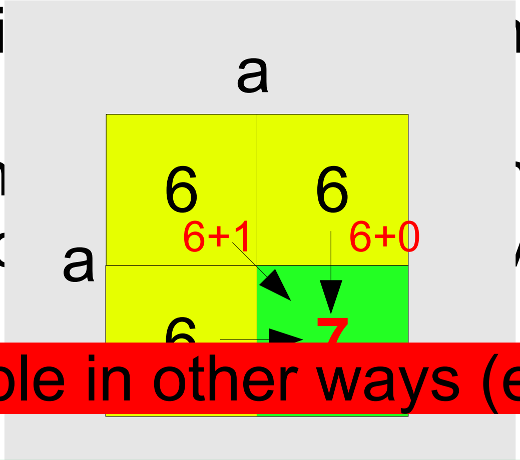
	a	c	g	t
a	1	0	0	0
c	0	1	0	0
[illegible]	[illegible]	[illegible]	1	0
t	0	0	0	1

gapcost: 0

Basic idea: Filling out a table



The algorithm essentially fi … amic programming)



Observation: The value in … three of its neighbor cells (left, upper-… ve) ...

Score matrix:

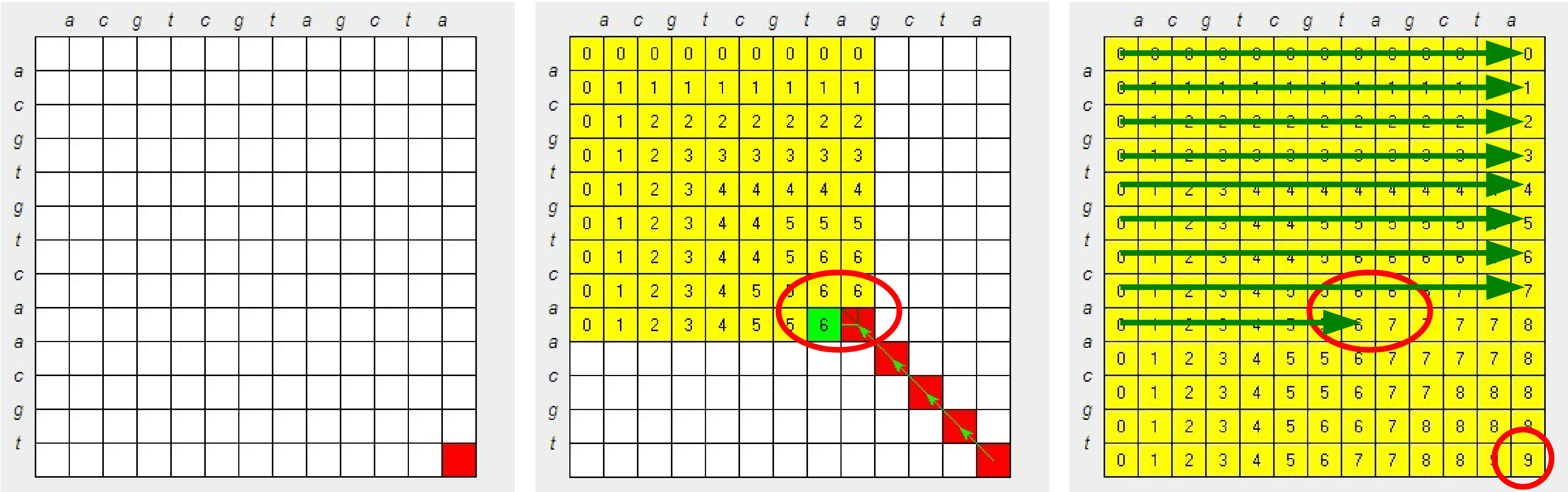
	a	c	g	t
a	1	0	0	0
c	0	1	0	0
			1	0
t	0	0	0	1

…st: 0

Can we fill out the table in other ways (e.g. non-recursively)?

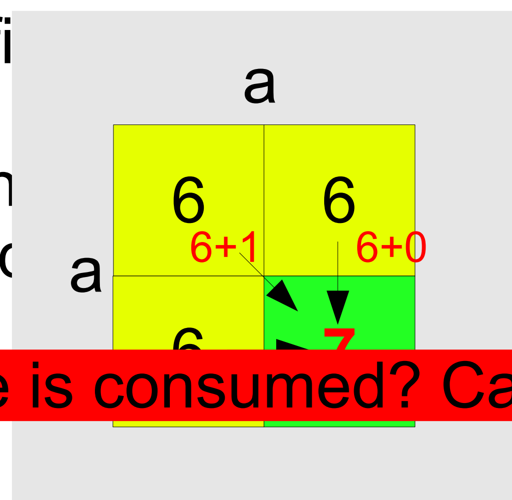
Row by row, or column by column, or diagonal by diagonal

Basic idea: Filling out a table



The algorithm essentially fi   amic programming)

Observation: The value in   three of its
neighbor cells (left, upper-   ve) ...



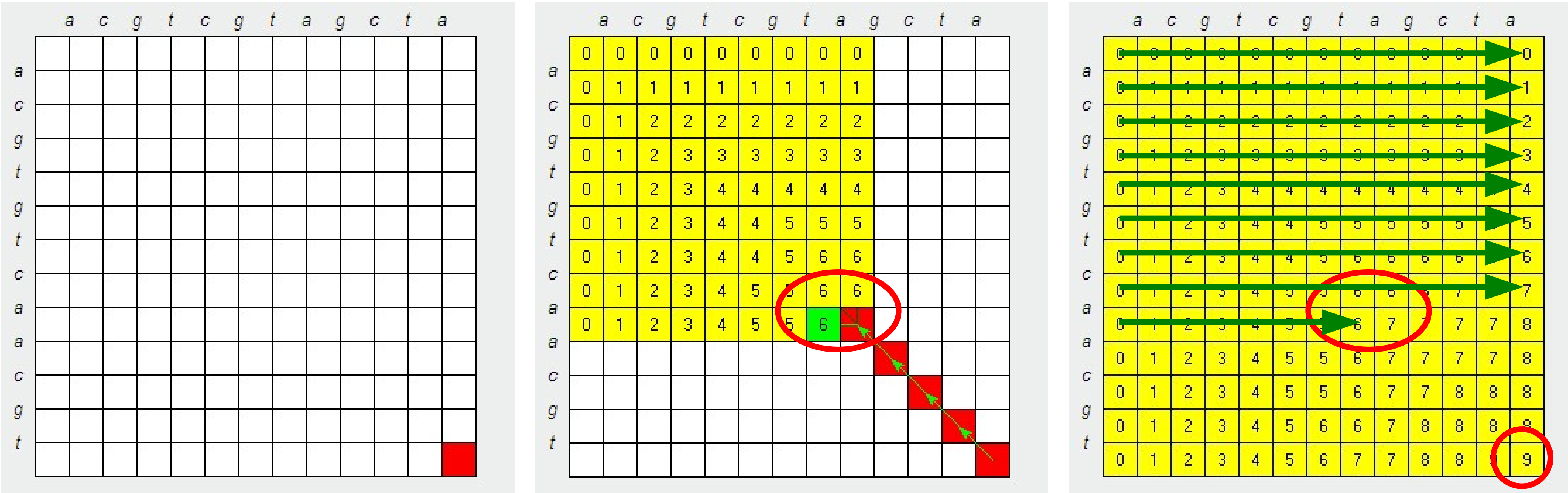
How much space is consumed? Can it be improved?

Score matrix:

	a	c	g	t
a	1	0	0	0
c	0	1	0	0
g	0	0	1	0
t	0	0	0	1

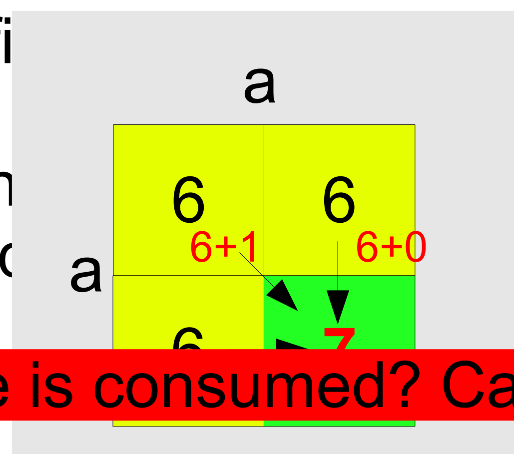
gapcost: 0

Basic idea: Filling out a table



The algorithm essentially fi ... ıamic programming)

Observation: The value in ... three of its neighbor cells (left, upper-... ve) ...



Score matrix:

	a	c	g	t
a	1	0	0	0
c	0	1	0	0
g	0	0	1	0
t	0	0	0	1

How much space is consumed? Can it be improved?

$O(n^2)$ can be improved to $O(n)$ by keeping only two rows in memory

Exercise

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

		a	c	t	a
	0	-1			
a					
g					
g					
t					

Exercise

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

		a	c	t	a
	0	-1	-2		
a					
g					
g					
t					

Exercise

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

```
Score matrix:
   a c g t
 a 2 0 0 0
 c 0 2 0 0
 g 0 0 2 0
 t 0 0 0 2
```

```
gapcost: -1
```

		a	c	t	a
	0	-1	-2	-3	
a					
g					
g					
t					

Exercise

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

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

Exercise

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

```
Score matrix:
   a c g t
 a 2 0 0 0
 c 0 2 0 0
 g 0 0 2 0
 t 0 0 0 2
```

```
gapcost: -1
```

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

Exercise

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

```
Score matrix:
    a c g t
  a 2 0 0 0
  c 0 2 0 0
  g 0 0 2 0
  t 0 0 0 2
```

```
gapcost: -1
```

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

Exercise

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

```
Score matrix:
   a c g t
 a 2 0 0 0
 c 0 2 0 0
 g 0 0 2 0
 t 0 0 0 2
```

```
gapcost: -1
```

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

Exercise

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

```
Score matrix:
    a c g t
  a 2 0 0 0
  c 0 2 0 0
  g 0 0 2 0
  t 0 0 0 2
```

```
gapcost: -1
```

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

Exercise

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

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

Exercise

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

```
Score matrix:
    a c g t
  a 2 0 0 0
  c 0 2 0 0
  g 0 0 2 0
  t 0 0 0 2
```

```
gapcost: -1
```

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

Exercise

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

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

Exercise

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

```
Score matrix:
    a c g t
  a 2 0 0 0
  c 0 2 0 0
  g 0 0 2 0
  t 0 0 0 2
```

```
gapcost: -1
```

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

Exercise

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

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

Exercise

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

```
Score matrix:
    a c g t
  a 2 0 0 0
  c 0 2 0 0
  g 0 0 2 0
  t 0 0 0 2
```

```
gapcost: -1
```

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

Exercise

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

```
Score matrix:
    a c g t
  a 2 0 0 0
  c 0 2 0 0
  g 0 0 2 0
  t 0 0 0 2
```

```
gapcost: -1
```

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

Exercise

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

```
Score matrix:
   a c g t
 a 2 0 0 0
 c 0 2 0 0
 g 0 0 2 0
 t 0 0 0 2
```

```
gapcost: -1
```

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

Exercise

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

```
Score matrix:
    a c g t
  a 2 0 0 0
  c 0 2 0 0
  g 0 0 2 0
  t 0 0 0 2
```

```
gapcost: -1
```

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

Exercise

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

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

Exercise

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

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

Exercise

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

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

Exercise

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

```
Score matrix:
    a c g t
  a 2 0 0 0
  c 0 2 0 0
  g 0 0 2 0
  t 0 0 0 2
```

```
gapcost: -1
```

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

Exercise

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

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

Exercise

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

```
Score matrix:
    a c g t
  a 2 0 0 0
  c 0 2 0 0
  g 0 0 2 0
  t 0 0 0 2
```

```
gapcost: -1
```

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

Exercise

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

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

Exercise

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

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

The optimal score, but what about an optimal alignment?

Backtracking

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

```
Score matrix:
    a c g t
  a 2 0 0 0
  c 0 2 0 0
  g 0 0 2 0
  t 0 0 0 2
```

```
gapcost: -1
```

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

We find an optimal alignment by deciding which choice of columns has resulted in the optimal cost (lower right entry).

Backtracking

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

```
Score matrix:
    a c g t
  a 2 0 0 0
  c 0 2 0 0
  g 0 0 2 0
  t 0 0 0 2
```

```
gapcost: -1
```

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

```
-
a
```

We find an optimal alignment by deciding which choice of columns has resulted in the optimal cost (lower right entry).

Backtracking

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

```
Score matrix:
    a c g t
  a 2 0 0 0
  c 0 2 0 0
  g 0 0 2 0
  t 0 0 0 2
```

```
gapcost: -1
```

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

```
t -
t a
```

We find an optimal alignment by deciding which choice of columns has resulted in the optimal cost (lower right entry).

Backtracking

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

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

```
g t -
- t a
```

We find an optimal alignment by deciding which choice of columns has resulted in the optimal cost (lower right entry).

Backtracking

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

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

```
g g t -
c - t a
```

We find an optimal alignment by deciding which choice of columns has resulted in the optimal cost (lower right entry).

Backtracking

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

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

```
a g g t -
a c - t a
```

We find an optimal alignment by deciding which choice of columns has resulted in the optimal cost (lower right entry).

Backtracking

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

Also optimal:

```
a g g t -
a - c t a
```

```
a g g t
a c t a
```

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

```
a g g t -
a c - t a
```

We find an optimal alignment by deciding which choice of columns has resulted in the optimal cost (lower right entry).

Backtracking

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

Also optimal:

```
a g g t -
a - c t a
```

```
a g g t
a c t a
```

		a	c	t	a
	0	-1	-2	-3	-4
a	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
g	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
g	-3	0	1	2	1
t	-4	-1	0	3	2

Running time?

Space consumption?

```
a g g t -
a c - t a
```

We find an optimal alignment by deciding which choice of columns has resulted in the optimal cost (lower right entry).

Backtracking

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

Score matrix:

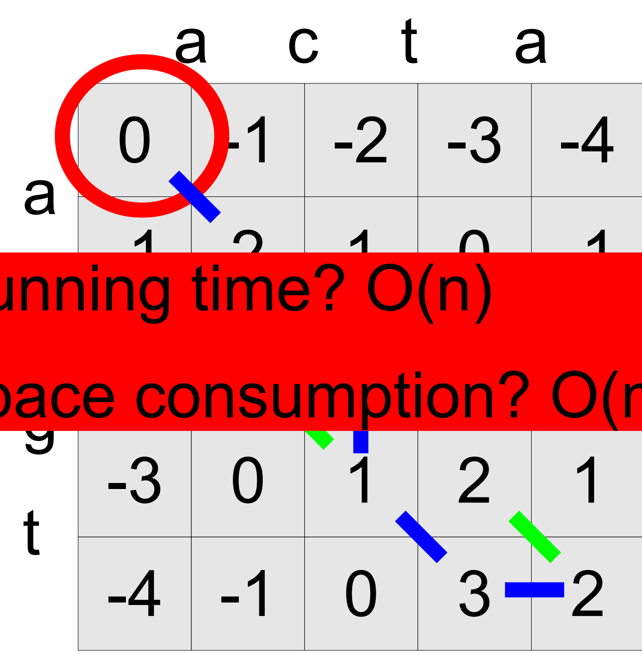
	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

Also optimal:

```
a g g t -
a - c t a
```

```
a g g t
a c t a
```



Running time? O(n)

Space consumption? O(nm)

```
a g g t -
a c - t a
```

We find an optimal alignment by deciding which choice of columns has resulted in the optimal cost (lower right entry).

Backtracking

Compute the optimal score of an optimal alignment of **aggt** and **acta** using:

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

Also optimal:

```
a g g t -
a - c t a
```

```
a g g t
a c t a
```

		a	c	t	a
	0	-1	-2	-3	-4
a	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
g	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
g	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
t	-4	-1	0	3	2

Running time? O(n)

Space consumption? O(nm)

```
a g g t -
a c - t a
```

Another lecture: Space consumption can be improve to O(n)

We find an optimal alignment by deciding which choice of columns has resulted in the optimal cost (lower right entry).

Backtracking

```
func Cost(i, j):
    if T[i,j] = undef then
        v1 = v2 = v3 = v4 = undef
        if (i > 0) and (j > 0) then
            v1 = Cost(i-1, j-1) + d(A[i],B[j])
        if (i > 0) and (j >= 0) then
            v2 = Cost(i-1, j) + g
        if (i >= 0) and (j > 0) then
            v3 = Cost(i, j-1) + g
        if (i = 0) and (j = 0) then
            v4 = 0
        T[i,j] = max(v1,v2,v3,v4)
    endif
    return T[i,j]
end
T[0..n][0..m]=undef
print Cost(n,m)
```

an optimal
sing:

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

			t	a
			-3	-4
			0	1
3	0	1	2	1
-4	-1	0	3	2

O(n)

Space consumption? O(nm)

```
a g g t -
a c - t a
```

a c t a

Another lecture: Space consumption can be improve to O(n)

We find an optimal alignment by deciding which choice of columns has resulted in the optimal cost (lower right entry).

Backtracking

```
func Cost(i
    if T[i,
        v1
        if

        if

        if

        if

        T[i
    endif
    return
end
T[0..n][0..m]=undef
print Cost(n,m)
```

```
func RecurBackTrack(i, j):
    if (i > 0) and (j > 0) and T[i,j] == T[i-1, j-1] + subcost(A[i], B[j]) then
        "output column(A[i], B[j])"
        RecurBackTrack(i-1, j-1)
    else if (i > 0) and (j > =0) and T[i,j] == T[i-1,j] + g then
        "output column(A[i], -)"
        RecurBackTrack(i-1, j)
    else if (i>=0) and (j > 0) and T[i,j] == T[i,j-1] + g then
        "output column(-, B[j])"
        RecurBackTrack(i, j-1)
    endif
end

RecurBackTrack(n,m)
```

O(n)

Space consumption? O(nm)

```
a g g t -
a c - t a
```

a c t a

Another lecture: Space consumption can be improve to O(n)

-4	-1	0	3	2

We find an optimal alignment by deciding which choice of columns has resulted in the optimal cost (lower right entry).

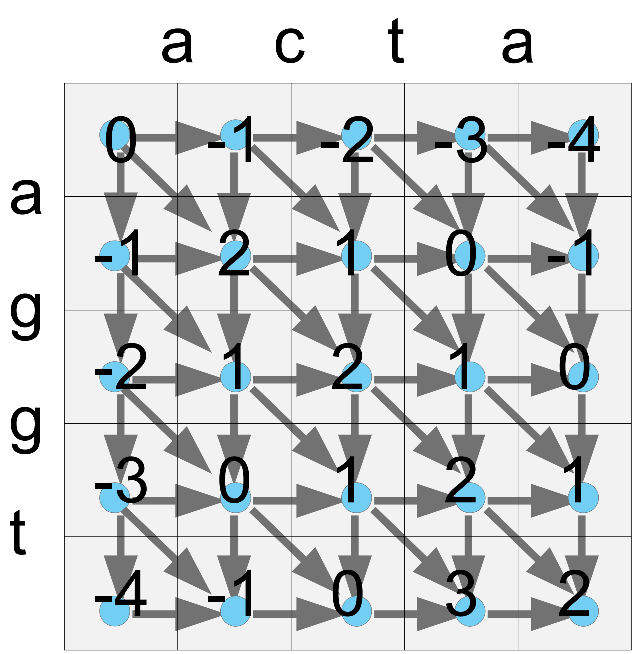
Alignment graph

An oriented weight (grid) graph where nodes “are” the cells in the dynamic programming matrix and edges “are” the recursive dependencies. The weight of an edge “is” the cost of the corresponding “alignment column” ...

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1



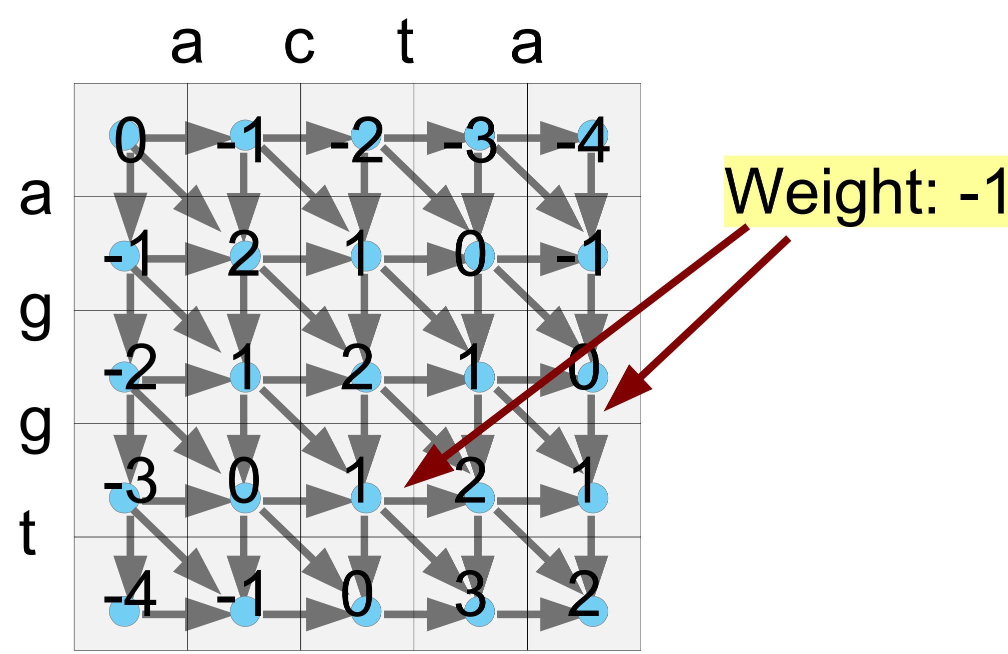
Alignment graph

An oriented weight (grid) graph where nodes “are” the cells in the dynamic programming matrix and edges “are” the recursive dependencies. The weight of an edge “is” the cost of the corresponding “alignment column” ...

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1



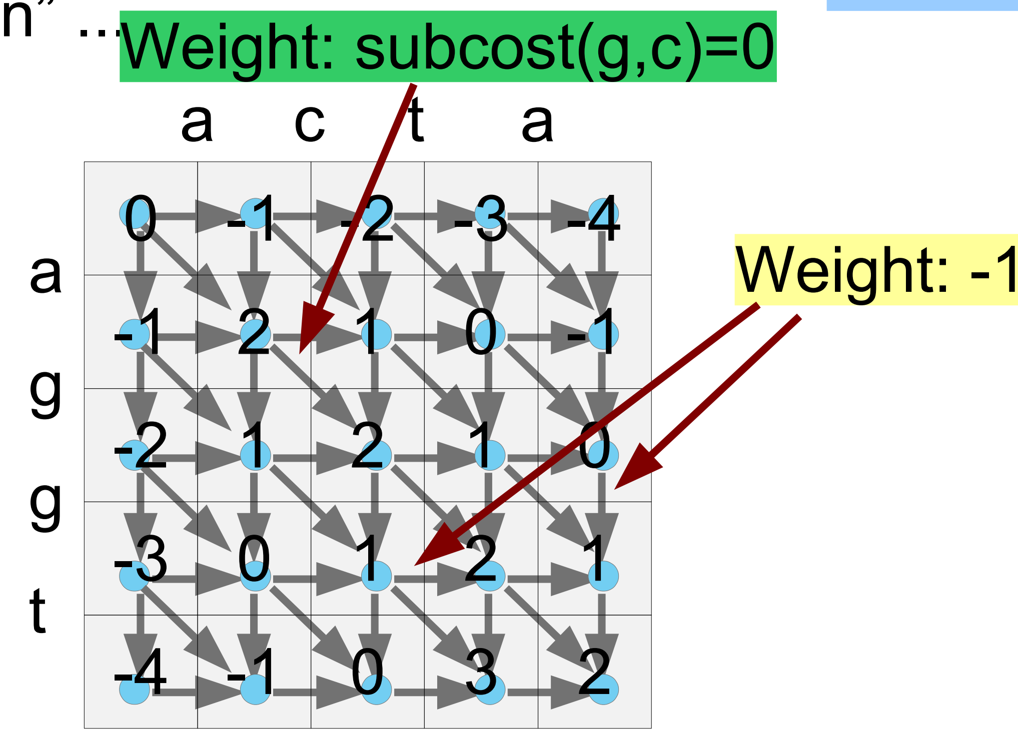
Alignment graph

An oriented weight (grid) graph where nodes “are” the cells in the dynamic programming matrix and edges “are” the recursive dependencies. The weight of an edge “is” the cost of the corresponding “alignment column” ...

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1



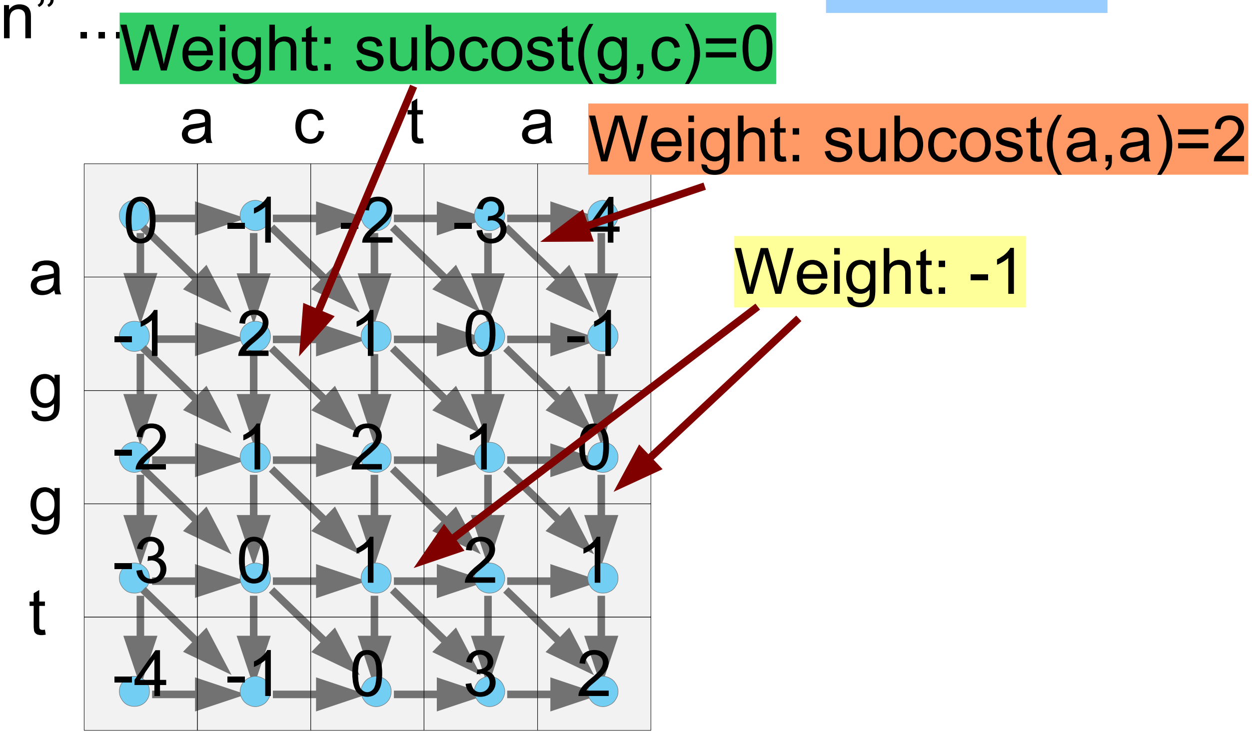
Alignment graph

An oriented weight (grid) graph where nodes “are” the cells in the dynamic programming matrix and edges “are” the recursive dependencies. The weight of an edge “is” the cost of the corresponding “alignment column” ...

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1

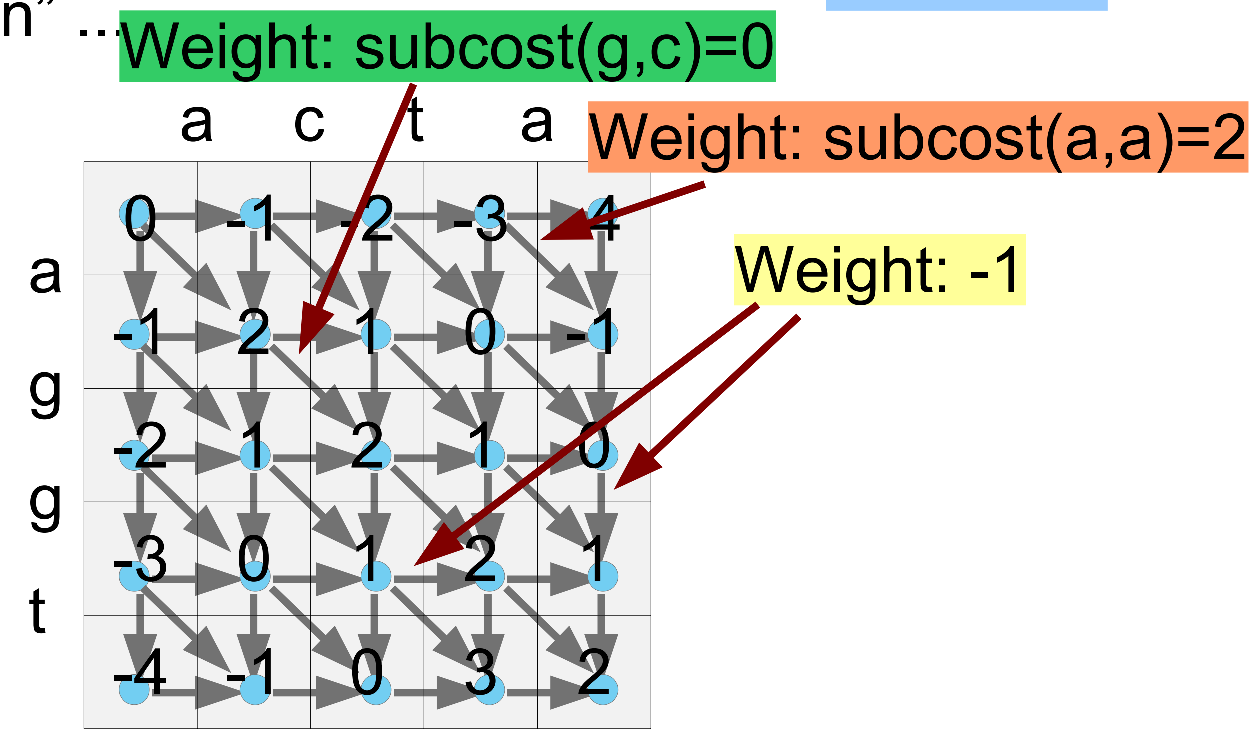


Alignment graph

An oriented weight (grid) graph where nodes “are” the cells in the dynamic programming matrix and edges “are” the recursive dependencies. The weight of an edge “is” the cost of the corresponding “alignment column” ...

```
Score matrix:
    a c g t
  a 2 0 0 0
  c 0 2 0 0
  g 0 0 2 0
  t 0 0 0 2
```

```
gapcost: -1
```



The cost of an optimal alignment is the length of a longest (shortest) path from (0,0) to (n,m), and the path yields the alignment ...

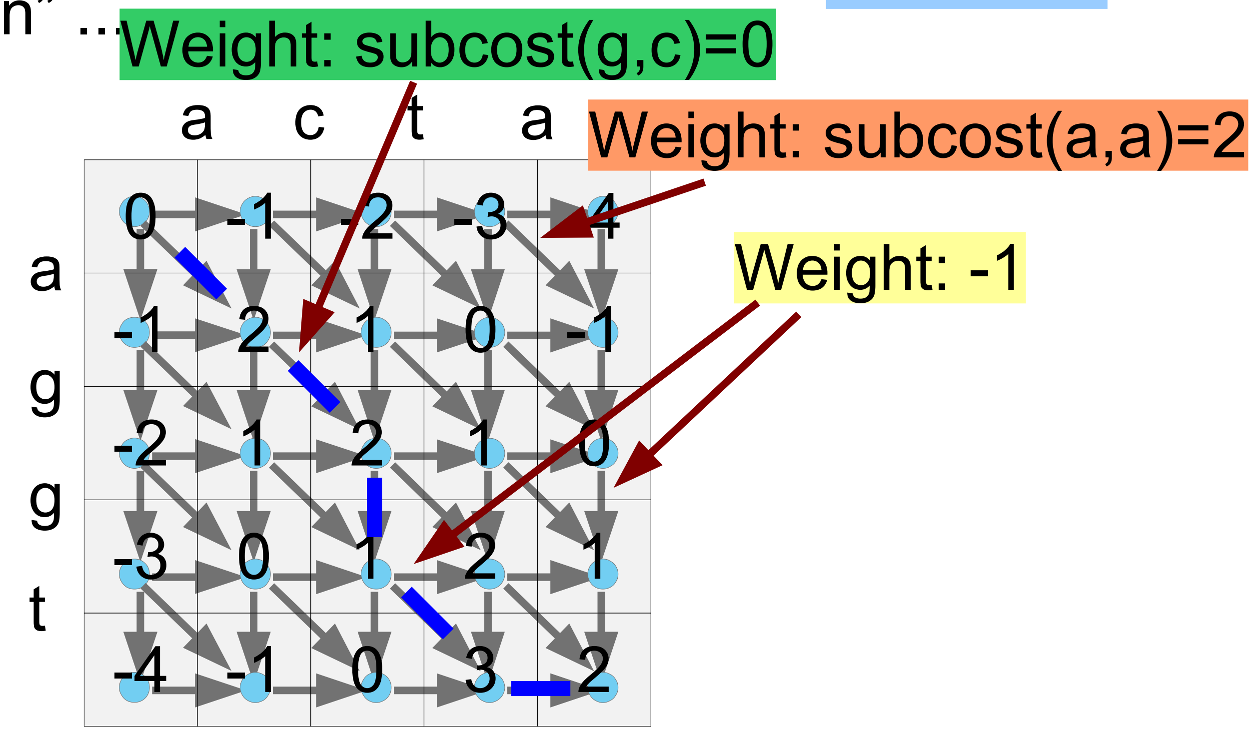
Alignment graph

An oriented weight (grid) graph where nodes “are” the cells in the dynamic programming matrix and edges “are” the recursive dependencies. The weight of an edge “is” the cost of the corresponding “alignment column” ...

Score matrix:

	a	c	g	t
a	2	0	0	0
c	0	2	0	0
g	0	0	2	0
t	0	0	0	2

gapcost: -1



The cost of an optimal alignment is the length of a longest (shortest) path from (0,0) to (n,m), and the path yields the alignment ...

Summary

You have been introduced to the notation of a pairwise alignment an algorithms for computing an optimal (global) pairwise alignment with linear gap cost.

You should be able to explain how to compute the cost of an optimal alignment in time $O(n^2)$ and how to find an optimal alignment by backtracking in time $O(n)$.

Summary

```
func Cost(i, j):
    if T[i,j] = undef then
        v1 = v2 = v3 = v4 = undef
        if (i > 0) and (j > 0) then
            v1 = Cost(i-1, j-1) + d(A[i],B[j])
        if (i > 0) and (j >= 0) then
            v2 = Cost(i-1, j) + g
        if (i >= 0) and (j > 0) then
            v3 = Cost(i, j-1) + g
        if (i = 0) and (j = 0) then
            v4 = 0
        T[i,j] = max(v1,v2,v3,v4)
    endif
    return T[i,j]
end
T[0..n][0..m]=undef
print Cost(n,m)
```

Input:

Sequences *A* and *B*, and score function ...

The column-score is given as substitution matrix *d* and a gap cost *g* ...

	A	C	G	T
A	10	2	5	2
C	2	10	2	5
G	5	2	10	2
T	2	5	2	10

gapcost: -5

Summary

```
func Cost(i, j):
    if T[i,j] = undef then
        v1 = v2 = v3 = v4 = undef
        if (i > 0) and (j > 0) then
            v1 = Cost(i-1, j-1) + d(A[i],B[j])
        if (i > 0) and (j >= 0) then
            v2 = Cost(i-1, j) + g
        if (i >= 0) and (j > 0) then
            v3 = Cost(i, j-1) + g
        if (i = 0) and (j = 0) then
            v4 = 0
        T[i,j] = max(v1,v2,v3,v4)
    endif
    return T[i,j]
end
T[0..n][0..m]=undef
print Cost(n,m)
```

Input:

Sequences *A* and *B*, and score function ...

The column-score is given as substitution matrix *d* and a gap cost *g* ...

	A	C	G	T
A	10	2	5	2
C	2	10	2	5
G	5	2	10	2
T	2	5	2	10

gapcost: -5

Transitions (mutations between two purines (A,G) or two pyrimidines (T,C)) are more likely than transversions (mutations between a purine and a pyrimidine) ...

Summary

```
func Cost(i, j):
    if T[i,j] = undef then
        v1 = v2 = v3 = v4 = undef
        if (i > 0) and (j > 0) then
            v1 = Cost(i-1, j-1) + d(A[i],B[j])
        if (i > 0) and (j >= 0) then
            v2 = Cost(i-1, j) + g
        if (i >= 0) and (j > 0) then
            v3 = Cost(i, j-1) + g
        if (i = 0) and (j = 0) then
            v4 = 0
        T[i,j] = max(v1,v2,v3,v4)
    endif
    return T[i,j]
end
T[0..n][0..m]=undef
print Cost(n,m)
```

Input:

Sequences *A* and *B*, and score function ...

The column-score is given as substitution matrix *d* and a gap cost *g* ...

	A	C	G	T
A	10	2	5	2
C	2	10	2	5
G	5	2	10	2
T	2	5	2	10

gapcost: -5

Note: Can be implemented as “maximizing a similarity”, as above, or “minimizing a cost (or distance)”, with “min” instead of “max” ...

Summary

```
func Cost(i, j):
    if T[i,j] = undef then
        v1 = v2 = v3 = v4 = undef
        if (i > 0) and (j > 0) then
            v1 = Cost(i-1, j-1) + d(A[i],B[j])
        if (i > 0) and (j >= 0) then
            v2 = Cost(i-1, j) + g
        if (i >= 0) and (j > 0) then
            v3 = Cost(i, j-1) + g
        if (i = 0) and (j = 0) then
            v4 = 0
        T[i,j] = max(v1,v2,v3,v4)
    endif
    return T[i,j]
end
T[0..n][0..m]=undef
print Cost(n,m)
```

Input:

Sequences *A* and *B*, and score function ...

The column-score is given as substitution matrix *d* and a gap cost *g* ...

	A	C	G	T
A	10	2	5	2
C	2	10	2	5
G	5	2	10	2
T	2	5	2	10

gapcost: -5

Can e.g. be used to compute: (1) Unit cost edit-distance, or (2) length of longest common subsequence. How?

Extension – Modeling gapcost

Biological observation: longer insertions and deletions (indels) are more common than shorter indels, i.e. a "good" alignment tends to few long indels rather than many short indels ...

Can the simple algorithm for pairwise alignment be adapted to reflect this additional biological insigth, i.e. a better model of biology?

Yes, we introduce the concept of a gapcost-function g(*k*) which gives the cost/penalty for a block of *k* consecutive insertions or deletions ...

Example

```
A T A C A - - - C G C A
A T - C T C C A C - C T
```

s(A,A) + s(T,T) + g(1) + s(C,C) + s(A,T) + g(3) + s(C,C) + g(1) + s(C,C) + s(A,T)

Next time

Topics

Discussion of exercises

Handling general and affine gap cost

Local alignment, finding the most similar pair of substrings