Review:

① DNA.

② alignment with free software (Clustal omega)

③ edit distance

④ LCS

⑤ alignment - to maximize total of the column score.

Last column of an alignment

- Suppose we are to align S[1..i] and T[1..j]. Consider the last column of the optimal alignment. Three cases can happen:

$D[i,j]$

S[1..i-1]	S[i]	S[1..i-1]	S[i]	S[1..i]	-
T[1..j-1]	T[j]	T[1..j]	-	T[1..j-1]	T[j]
(1)		(2)		(3)	

$D[i-1,j-1]$ $f(S[i],T[j])$

- In each case, the sub-alignment without the last column is an optimal one (why?)

$$D[i,j] = D[i-1,j-1] + f(S[i],T[j])$$

S[1..i-1]	S[i]	S[1..i-1]	S[i]	S[1..i]	-
T[1..j-1]	T[j]	T[1..j]	-	T[1..j-1]	T[j]
✓ (1)		(2)		(3)	

$$D[i,j] = D[i-1,j] + f(S[i],-)$$

$$D[i,j] = D[i,j-1] + f(-,T[j]),$$

Recurrence Relation

- Denote the optimal alignment score of S[1..i], T[1..j] by D[i,j]. Then D[m,n] is the optimal alignment score.
- Let f(a,b) be the score between two letters a and b.
- Consider last column of the alignment.

$$\boxed{\begin{matrix}S[1..i-1]\\T[1..j-1]\end{matrix}}\quad\begin{matrix}S[i]\\T[j]\end{matrix}$$

- Case 1: S[i] v.s. T[j]
  - D[i,j] = D[i-1, j-1] + f(S[i], T[j]);

$$\boxed{\begin{matrix}S[1..i-1]\\T[1..j]\end{matrix}}\quad\begin{matrix}S[i]\\-\end{matrix}$$

- Case 2: S[i] v.s. -
  - D[i,j] = D[i-1, j] + f(S[i], -);

$$\boxed{\begin{matrix}S[1..i]\\T[1..j-1]\end{matrix}}\quad\begin{matrix}-\\T[j]\end{matrix}$$

- Case 3: - v.s. T[j]
  - D[i,j] = D[i, j-1] + f(-, T[j]);
- Therefore…

$$D[i,j] = \max\begin{cases} D[i-1, j-1] + f(S[i], T[j]); \\ D[i-1, j] + f(S[i], -); \\ D[i, j-1] + f(-, T[j]); \end{cases}$$

Algorithm

$|S| = m, |T| = n.$

Time complexity: $O(m, n)$.

```
D[0,0] = 0;
for i from 1 to m                         S[1..i] v.s ''
        D[i,0] = i* indel;                O(m)
for j from 1 to n
        D[0,j] = j* indel;                O(n)
for i from 1 to m                       }
        for j from 1 to n               } O(m·n)
                    ⎧ D[i-1, j-1] + f(S[i], T[j]);  |
     D[i,j] = max   ⎨ D[i-1, j] + f(S[i], -);       | O(1)
                    ⎩ D[i, j-1] + f(-, T[j]);       |
Output D[m,n];
```

Dynamic Programming Table

match = 1
mismatch = -1
indel = -1.

D[i, j]

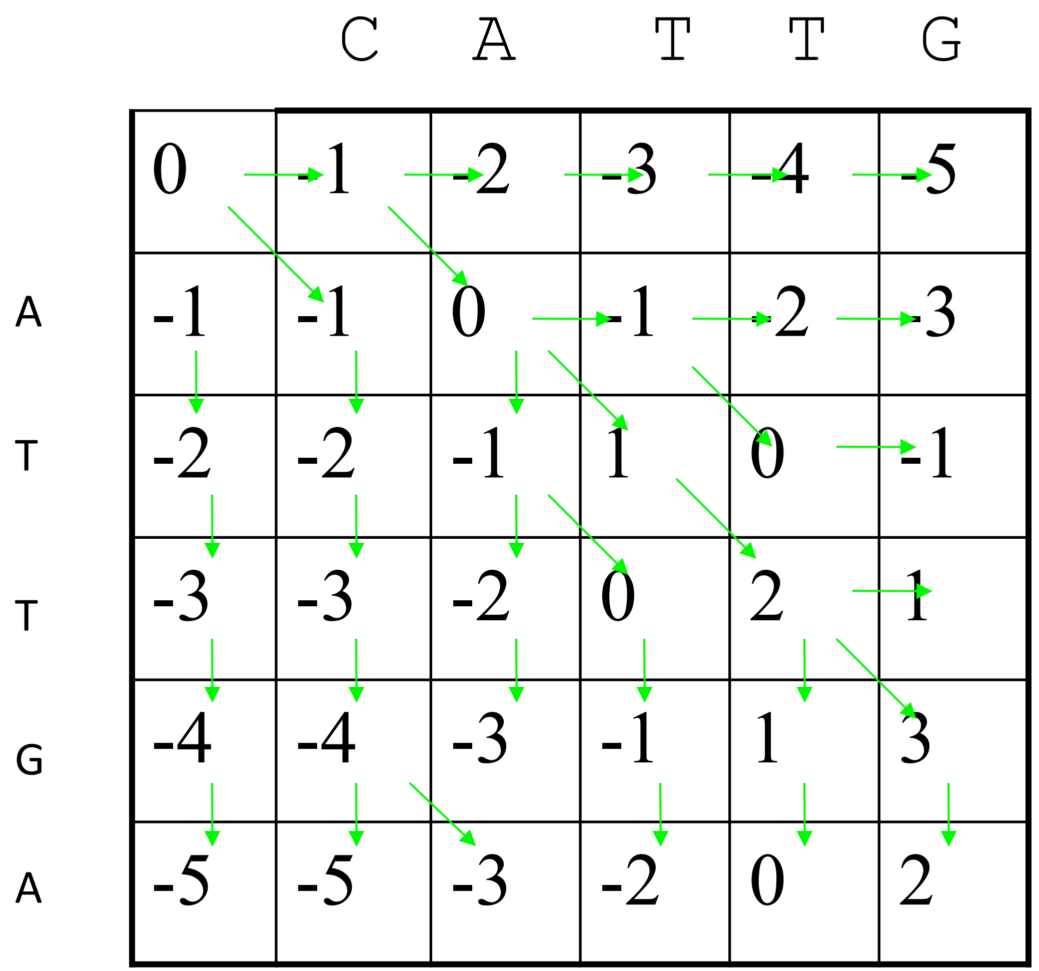
T → j

S ↓ i

		C	A	T	T	G
	0	-1	-2	-3	-4	-5
A	-1	-1	0	-1	-2	-3
T	-2	-2	-1	-2 -2		
T	-3					
G	-4					
A	-5					

$$D[i,j] = \max \begin{cases} D[i-1, j-1] + f(S[i], T[j]); \\ D[i-1, j] + f(S[i], -); \\ D[i, j-1] + f(-, T[j]); \end{cases}$$

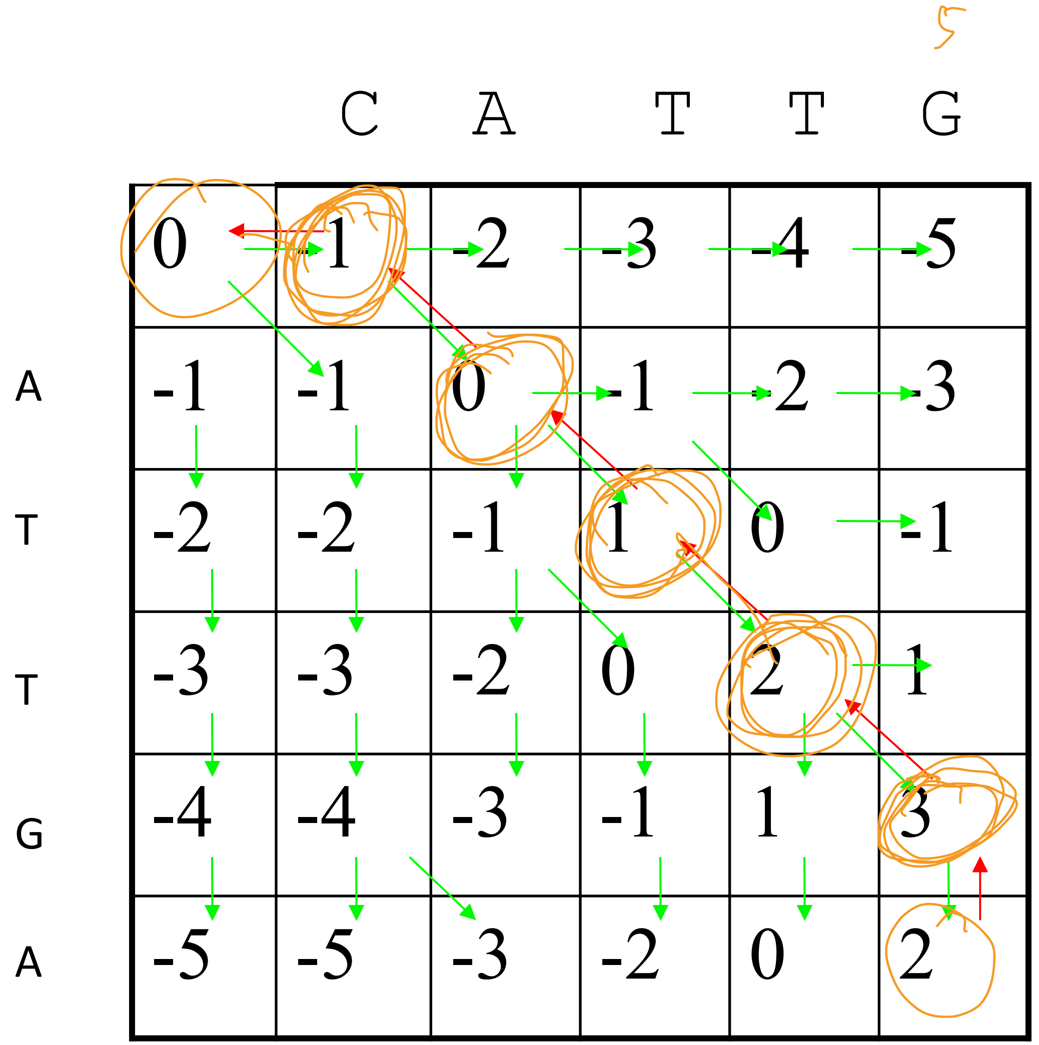
Dynamic Programming Table



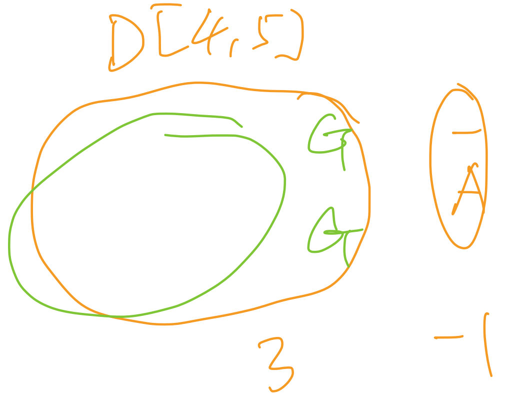
		C	A	T	T	G
	0	-1	-2	-3	-4	-5
A	-1	-1	0	-1	-2	-3
T	-2	-2	-1	1	0	-1
T	-3	-3	-2	0	2	1
G	-4	-4	-3	-1	1	3
A	-5	-5	-3	-2	0	2

$D[5,5] = D[4,5] + f(A,-)$.

Getting the actual alignment – backtracking



$$\begin{array}{c|cccccc}
 & & C & A & T & T & \overset{5}{G} \\
\hline
 & 0 & -1 & -2 & -3 & -4 & -5 \\
A & -1 & -1 & 0 & -1 & -2 & -3 \\
T & -2 & -2 & -1 & 1 & 0 & -1 \\
T & -3 & -3 & -2 & 0 & 2 & 1 \\
G & -4 & -4 & -3 & -1 & 1 & 3 \\
5\ A & -5 & -5 & -3 & -2 & 0 & 2
\end{array}$$



C A T T G –
– A T T G A

CATTG-
-ATTGA

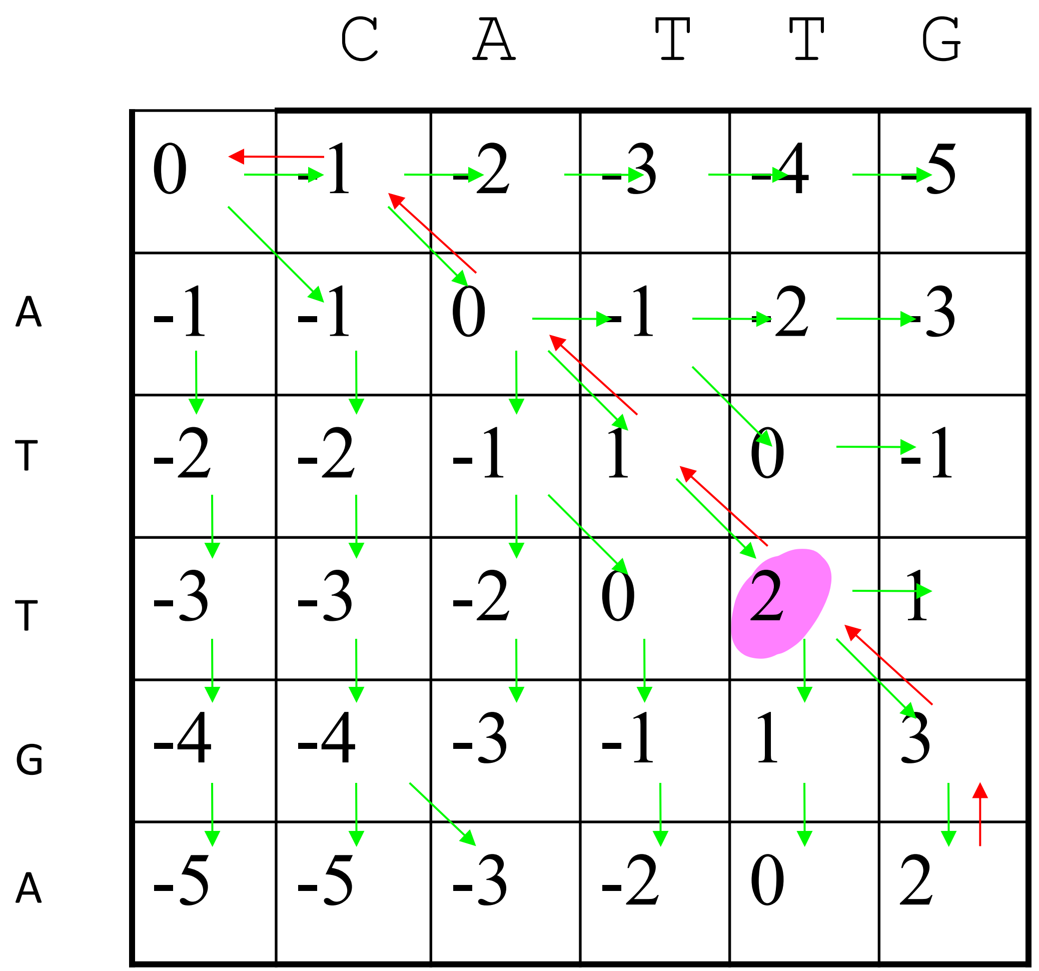
Complexity

- **Time Complexity**:
  - Filling the table takes O(nm) time: Each step requires only 3 checks to other points in the matrix.
  - How about the backtracking?
- Space Complexity:
  - O(nm)

A Practical Trick

D[3,4]

$$D[i,j] = \max \begin{cases} D[i-1,j-1] + f(S[i],T[j]) \\ D[i-1,j] + \text{indel} \\ D[i,j-1] + \text{indel}. \end{cases}$$



	C	A	T	T	G
	0	-1	-2	-3	-4	-5
A	-1	-1	0	-1	-2	-3
T	-2	-2	-1	1	0	-1
T	-3	-3	-2	0	2	1
G	-4	-4	-3	-1	1	3
A	-5	-5	-3	-2	0	2

O(m+n) steps

Added cost

No need to physically record the green arrows. Why?

O(m·n) writing of the arrows.

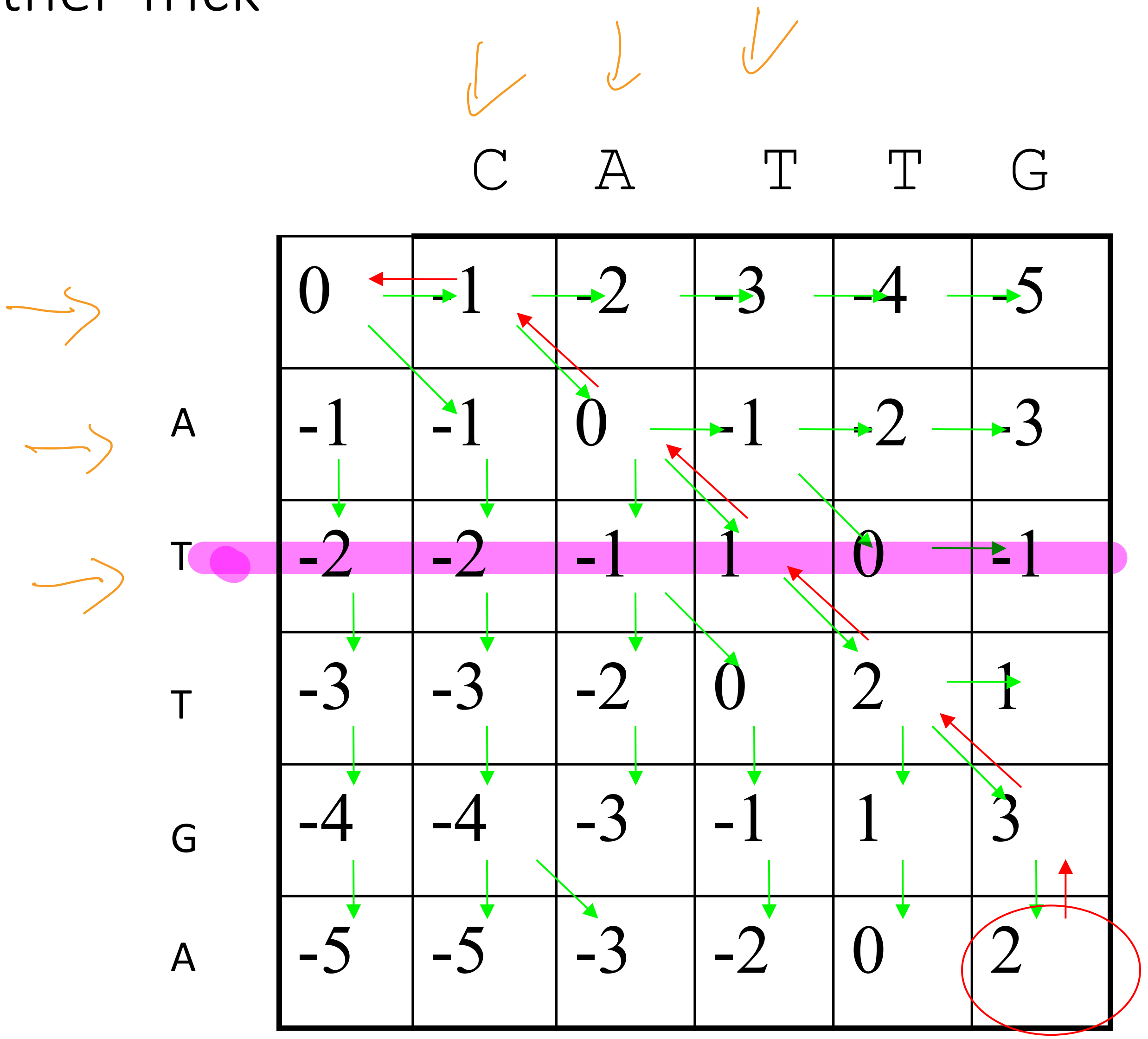
saved.

$\Rightarrow$ CATTG−
−ATTGA

Another Trick

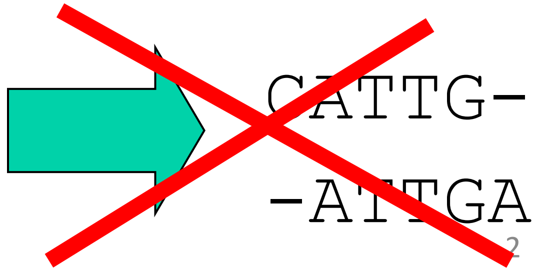
$O(n)$

$O(m)$ $O(\min(m,n))$



		C	A	T	T	G
	0	-1	-2	-3	-4	-5
A	-1	-1	0	-1	-2	-3
T	-2	-2	-1	1	0	-1
T	-3	-3	-2	0	2	1
G	-4	-4	-3	-1	1	3
A	-5	-5	-3	-2	0	2

If only score is needed, then space complexity can be reduced.



~~CATTG-~~

~~-ATTGA~~

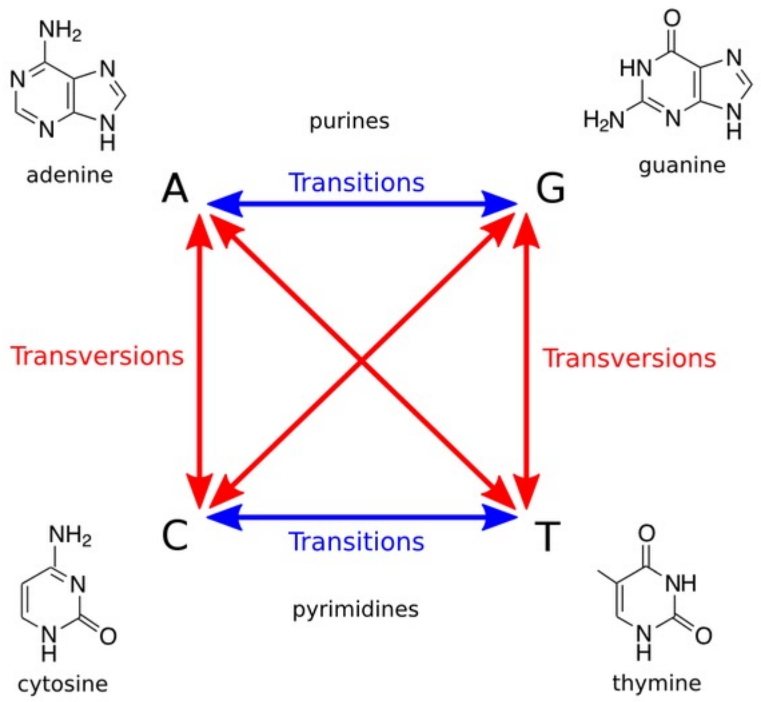
Score Function

- Now we have the algorithm for any score scheme f(x,y)
- Such separation of scoring and algorithm is a good thing. It allows us to optimize the score scheme independent to the algorithm.

“ The effective exploitation of his powers of abstraction must be regarded as one of the most vital activities of a competent programmer. ”

- Dijkstra

Transition vs. Transversion



- Transition happens more frequently; 2/3 of SNPs are transitions.
- In other words, transition is easier and therefore should be less penalized.

E.g.:

```
AAAGCAAA        AAAGCAAA
AAAT-AAA   vs   AAA-TAAA
```

- This can be easily achieved by changing score scheme f(a,b).

Alignment v.s. LCS vs. Edit Distance

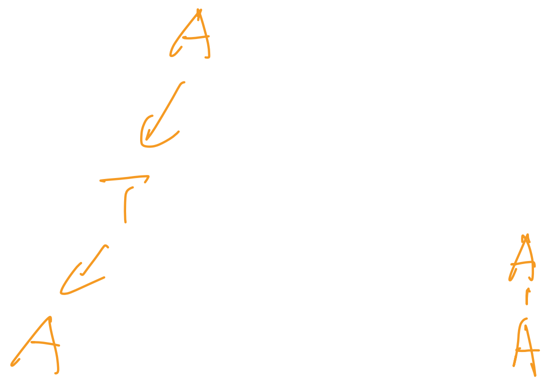
- By a properly defined score scheme, alignment can represent LCS and Edit distance, respectively.
  - match =
  - mismatch =
  - indel =

How to Build a Score Function

- First, know what you want.
- **Purpose 1**: the optimal alignment reveals the true evolutionary history.
- **Purpose 2**: high score indicates homology (derived from same ancestor).
- We want purpose 1 if possible, but purpose 2 is also useful.

```
ATGCA-TTTATTCCGAGG
||| | || ||| || ||
ATGTACTT-ATTACGTGG
```

Philosophy of a Score Function



- For purpose 1, right away: we might be **wrong**.
- That is, the alignment that has highest **score** may not be the one that actually matches evolutionary history.
- So you should never trust that an alignment must be right. It just optimizes the score.
- Should we give up purpose 1 at all?

Philosophy of A Score Function

- For purpose 1, the optimal alignment may be **approximately** correct **under certain conditions** in practice.
- As long as we know the limitation, we can still use it.
- For example, for the following alignment, it is “very likely” the alignment is approximately equal to the evolutionary history.
    - `ACGTATTACCGG-TTACCG`
    - `||| |||||||| ||||||`
    - `ACGGATTACCGGATTACCG`
- Limitation we keep in mind: when score is low, alignment itself is not too useful.

Gaps

```
AGATTTTTTTC        AGATTTTTTTTTC
AGA---TTTTC        AGA-T-T-T-T-C
```

g(3)   3*g(1)

- The left seems "simpler" than the right.
- Indeed, during evolution, indels are relatively rare. However, insertion or deletion a segment of k consecutive bases is much easier than k scattered indels.
- But our current scoring method (adding up column scores) cannot distinguish the two.
- Currently, a gap of length k costs k*indel. Thus, this is called the **linear gap penalty**.

S[1..i-k] S[i-k+1..i]

T[1..j] — — — — — —

$\underbrace{\qquad\qquad}_{k}$

Arbitrary gap penalty

- Consecutive insertions or deletions are called a gap. Suppose the gap penalty of a length k gap is g(k) instead of the simple c*k.
- Assume g(x)+g(y)<= g(x+y). (Otherwise does not serve the purpose of grouping indels.)
- Can the old DP still work?

$D[i,j] = D[i-k,j] + g(k)$

S[1..i-1]	S[i]	S[1..i-1]		S[i]	S[1..i]		-
T[1..j-1]	T[j]	T[1..j]		-	T[1..j-1]		T[j]
(1)		(2)			(3)		

$g(k) - g(k-1)$

Arbitrary Gap Penalty

- Old algorithm does not work anymore because we do not know the contribution of the last column to the gap penalty in the last two cases.
- The length of the gap is needed.

Alignment Algorithm for Arbitrary Gap Penalty

- We still use D[i,j] to denote the optimal alignment score of S[1..i] and T[1..j].
- We change cases 2 and 3 to include the last gap (not the last column).
- D[i,j] = max of the following three cases:
  - D[i-1,j-1]+f(s[i],t[j]).   (s[i] v.s. t[j])
  - $\max_{1 \le k \le i}$ D[i-k,j] + g(k)
  - $\max_{1 \le k \le j}$ D[i,j-k] + g(k)

D[i,j] = max

for i = 1..m
    for j = 1..n   } m×n

recurrence relation: O(m+n)

O(m×n×(m+n))

S[1..i-1]	S[i]	S[1..i-k]	S[i-k+1..$\overset{i}{\cancel{k}}$]	S[1..i]	-…-
T[1..j-1]	T[j]	T[1..j]	-…-	T[1..j-k]	T[j-k+1..$\underset{j}{\cancel{k}}$]
(1)		(2)		(3)	

39

Time Complexity

lecture 2022·01-13 stopped here.

- Cubic time complexity.
- In bioinformatics, very often we face the choice between:
  - Reality: How close it approximates the real biology
  - Simplicity: How easy it can be computed
- Now let’s simplify the g(k) a little. We basically want a function that grows slower than linear.
- g(k) = a + b*k
  - a = gap open penalty
  - b = gap extension penalty
- This is called **affine gap penalty,** in contrast to linear gap penalty**.**