## Sequence alignment

Algorithms:

Needleman and Wunsch Smith and Waterman

#### Edit distance

- Two sequences:
  - ACTTGTAGATG
  - ACTTGTAGATG
- Although the two sequences look different, one can change the first one to the second by three editing operations
  - insert "G" at the end of seq1
  - substitute the 8<sup>th</sup> letter "G" with "A"
  - delete the 3<sup>rd</sup> letter "A"

#### Edit distance

- The edit distance between two sequences is the minimum number of editing operations that need to convert one to the other.
- Editing operations:
  - insertion, deletion, substitution
- d(s, t) = d(t, s)
  - Proof: .....

## Weights of different operation

- Substitution may be easier than indel (insertion/deletion).
  - Give higher cost to indel
- Substitution between different pairs may be different.
- To satisfy d(s,t) = d(t,s)
  - cost(insertion) = cost(deletion)
  - $-\cos(substitute(x,y)) = \cos(substitute(y,x))$

## Sequence alignment

- Sequence comparison can be done with an alignment:
  - ACATTGTGGAT-
  - AC-TTGTAGATG
- In the above alignment, there are three mismatches (A, -), (G, A), (-,G)
- If match has score 1, mismatch has score -1, then the score of the alignment is 9-3=6.
- The (pairwise) sequence alignment is to find the optimal (with highest score) alignment for the two sequences.

#### Alignment v.s. edit distance

Let match score be

```
- f(x,y) = 0, if x==y

- f(x,y) = -cost(x,y), if x!=y

- f(-,x) = f(x,-) = -cost(indel)
```

- Then the optimal alignment score is equal to negative edit distance.
- Each edit distance problem can be reduced to a sequence alignment problem.

#### Formal definition

- Let s, t be two sequences over alphabet Σ.
   Let f(x,y) be a score scheme. An
   alignment is two sequences S, T with
   same length, generated by inserting
   spaces into s, t, respectively. The score of
   the alignment is sum(f(S[i],T[i])).
- The sequence alignment is to find the alignment with maximum score.

## Dynamic Programming

- Dynamic programming is a general approach to design algorithms in computer science.
- The idea is:
  - If a function f(n) can be computed using f(n-1), f(n-2), ..., f(1), then we can compute f(i) for i from 1 to n.

## Alignment

- Let len(s) = m and len(t) = n. Let DP[i,j] be the optimal alignment score for s[1..i] and t[1..j].
- Therefore, the optimal alignment score of s and t is DP[m,n].
- Now let's find out how to compute DP[i,j] from DP[a,b] for all a<i or b<j.</li>

#### Recursive definition

```
Case 1: s[i] matches t[j]
    - DP[i,j] = DP[i-1, j-1] + f(s[i], t[j]);
Case 2: s[i] matches -
    - DP[i,j] = DP[i-1, j] + f(s[i], -);
Case 3: t[i] matches -
    - DP[i,j] = DP[i, j-1] + f(-, t[j]);
• Therefore...
        DP[i,j] = \max \begin{cases} DP[i-1, j-1] + f(s[i], t[j]); \\ DP[i-1, j] + f(s[i], -); \\ DP[i, j-1] + f(-, t[j]); \end{cases}
```

#### Algorithm

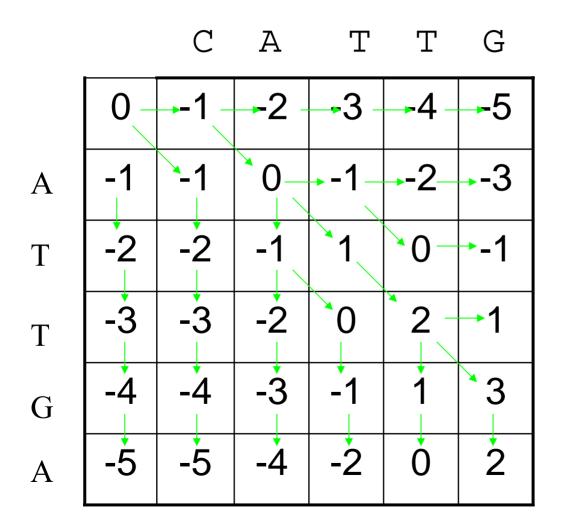
```
DP[0,0] = 0;
DP[i,0] = DP[i-1,0] + f(s[i],-);
DP[0,j] = DP[0,j-1] + f(-, t[j]);
for i from 1 to m
    for j from 1 to n
       DP[i,j] = \max \begin{cases} DP[i-1, j-1] + f(s[i], t[j]); \\ DP[i-1, j] + f(s[i], -); \\ DP[i, j-1] + f(-, t[j]); \end{cases}
Output DP[m,n];
```

## Figure

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

CATTG--ATTGA

# Getting the actual alignment – backtracking



CATTG-

#### complexity

- Time complexity O(mn)
- Space complexity O(mn)
- How to compute the alignment score with O(min(m,n)) space?
  - Hint: we do not need the actual alignment now.
- There is an algorithm that outputs the best alignment with O(min(m,n)) space and O(mn) time.

## Fit one sequence into the other

- Let s, t be two sequences. Find t[a..b] that aligns with s with the best alignment score.
- Let DP[i,j] be the best alignment score between s[1..i] and t[a..j].
- Then the best alignment score is max(DP[m,j]) for all j.

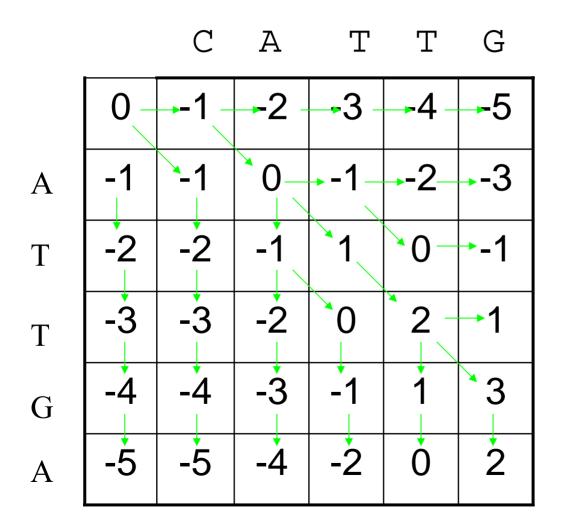
#### Recursive definition

- Case 1: s[i] matches t[j]
   DP[i,j] = DP[i-1, j-1] + f(s[i], t[j]);
- Case 2: s[i] matches -
  - DP[i,j] = DP[i-1, j] + f(s[i], -);
- Case 3: t[j] matches -
  - DP[i,j] = DP[i, j-1] + f(-, t[j]);
- Therefore  $\begin{aligned} & DP[i-1,j-1] + f(s[i],t[j]); \\ & DP[i,j] = \max \end{aligned} \begin{cases} & DP[i-1,j-1] + f(s[i],t[j]); \\ & DP[i-1,j] + f(s[i],-); \\ & DP[i,j-1] + f(-,t[j]); \end{aligned}$

#### Algorithm

```
DP[0,0] = 0;
DP[i,0] = DP[i-1,0] + f(s[i],-);
DP[0,j] = 0;
for i from 1 to m
    for j from 1 to n
       DP[i,j] = \max \begin{cases} DP[i-1, j-1] + f(s[i], t[j]); \\ DP[i-1, j] + f(s[i], -); \\ DP[i, j-1] + f(-, t[i]); \end{cases}
Output max { DP[m,j] \mid 0 \le j \le n };
```

# Getting the actual alignment – backtracking



CATTG-

## Local Alignment

- Given sequences s, t, find s[a..b] and t[c..d] such that the alignment score of s[a..b] and t[c..d] are the highest.
- Example:
  - AGA<mark>TTGTGG</mark>-TA
  - AC-<mark>TTG-GG</mark>ATG
  - The best local alignment should be TTGTGG v.s. TTGGG
- Trivial approach ......

#### A better method

- Let DP[i,j] be the optimal alignment score for s[a,i] and t[b, j] for all a<=i+1 and b<=j+1.</li>
  - Two prefixes of s[1,i] and t[1,j].
- Then the best local alignment score is max(DP[i,j]) for all i and j.

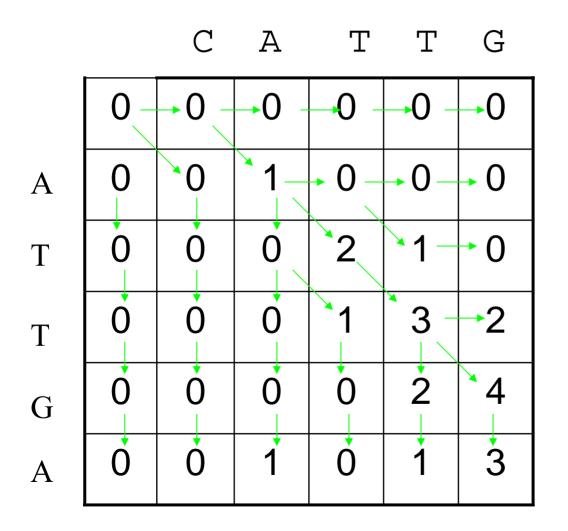
#### A better method

```
Case 1: s[i] matches t[j]
    - DP[i,j] = DP[i-1, j-1] + f(s[i], t[j]);
Case 2: s[i] matches -
    - DP[i,i] = DP[i-1, j] + f(s[i], -);
• Case 3: t[i] matches -
    - DP[i,j] = DP[i, j-1] + f(-, t[j]);
Case 4: DP[i,j]=0;
• Therefore...
                           DP[i-1, j-1] + f(s[i], t[j]);
DP[i-1, j] + f(s[i], -);
DP[i, j-1] + f(-, t[j]);
```

#### Algorithm

```
DP[0,0] = 0;
DP[i,0] = 0;
DP[0,j] = 0;
for i from 1 to m
   for j from 1 to n
                             DP[i-1, j-1] + f(s[i], t[j]);
DP[i-1, j] + f(s[i], -);
DP[i, j-1] + f(-, t[j]);
       DP[i,j] = max
Output max(DP[i,j]);
```

# Getting the actual alignment – backtracking



CATTG--ATTGA

## Gapped Alignment

- Which of the following two alignments is better?
  - -ACATTGTGGAT
  - -AC-T-GTAGAT

#### and

- -ACATTGTGGAT
- -AC--TGTAGAT

## General gap penalty

```
DP[i,j] = max \begin{cases} DP[i-1, j-1] + f(s[i], t[j]); \\ DP[i-g, j] + f(g), (g=1...i); \\ DP[i, j-g] + f(g), (g=1...i). \end{cases}
```

Time: O(m\*n\*max(m,n))

## Affined gap penalty

For a gap with k consecutive spaces, we use

$$g(k) = a + k*b$$

as the penalty. a is called gap open penalty, b is the gap extension

#### Recursive definition

- Let DP0[i,j] be the alignment score that s[i] matches t[j].
- Let DP1[i,j] be the alignment score that s[i] matches -.
- Let DP2[i,j] be the alignment score that t[j] matches -.

#### Recursive definition

$$DP0[i,j] = f(s[i], t[j]) + max \begin{cases} DP0[i-1, j-1]; \\ DP1[i-1, j-1]; \\ DP2[i-1, j-1]; \end{cases}$$

$$DP1[i,j] = f(s[i], -) + max \begin{cases} DP0[i-1, j] + a; \\ DP1[i-1, j]; \\ DP2[i-1, j] + a; \end{cases}$$

$$DP2[i,j] = f(s[i], t[j]) + max \begin{cases} DP0[i, j-1] + a; \\ DP1[i, j-1] + a; \\ DP2[i, j-1]; \end{cases}$$

#### Algorithm

No difference but use three arrays.