Alignments & Co

(approx. pattern matching)

Basics: Dynamic Programming (DP)

"programming" refers to "tabular" method, not to writing program code.

DP is a general, powerful alg. design technique for solving optimization problems

DP ≙ type of very smart exhaustive search that can be applied if the problem can be "subdivided" into overlapping subproblems.

Exmpl. Fibonacci numbers: $f(1) = f(2) = 1$

$f(n) = f(n-1) + f(n-2)$

⇒ 1, 1, 2, 3, 5, 8, 13, ...

Naive recursive way:

```
F(n)
  IF(n ≤ 2) f = 1
  ELSE  f = F(n-1) + F(n-2)
  return f
```

Exponential time!

Why: Recurrence Nr:

$T(n)$ = time to compute n-th Fibnr.

$\Rightarrow T(n) = T(n-1) + T(n-2) + O(1)$

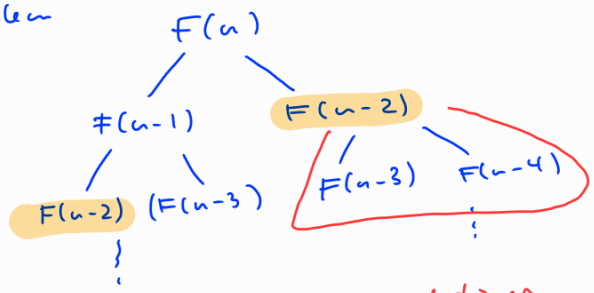
$\geq 2T(n-2)$

$\geq 2 \cdot 2\, T(n-4)$ ($T(n-1) \geq T(n-2)$)

$\vdots$

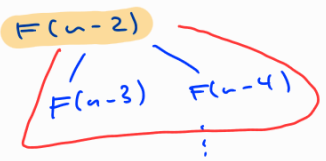
$\geq 2^{\frac{n}{2}} = O(2^n)$

"problem"



We compute F(k) several times, although it is enough to compute it ones

We can prune the entire search tree:

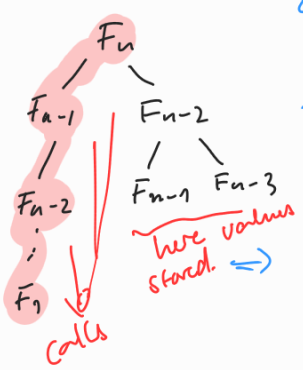


⇒ we must "memorize" the intermediate results.

```
memo = ∅        // dictionary

mF(n)
    IF (n ∈ memo) return memo[n]      // check if F[n] already computed.
    IF (n ≤ 2)  f = 1
    ELSE  f = mF(n-1) + mF(n-2)       } still the same lines.
    memo[n] = f
    return f
```

Runtime sketch: ⇒ mF(k) only recurses the first time it's called ∀ k & memoized calls cost O(1)



the nr of non-memoized calls is n:
mF(1), mF(2), ... mF(k) ... mF(n)

⇒ runtime O(n) ! (goes even faster)

```
iterativ_f(n)
    list f(n) = 1xn empty list
    f(1) = f(2) = 1
    FOR (i = 3...n)
        f(i) = f(i-1) + f(i-2)
    return f
```

The Longest Common Subsequence (LCS) problem

This is a classical problem in Bioinformatics, where one wants to understand how "close" DNA/protein sequences are.

= simply a string some alphabet.

There are several ways to address this problem. Here we consider one of them: LCS.

Def: Let $X = x_1 \cdots x_m$ & $Z = z_1 \cdots z_k$ be two strings (= sequences)

Z is a subsequence of X if

$\exists$ indices of X $i_1 \cdots i_k$ s.t. $i_1 < i_2 < \cdots < i_k$

& $z_j = x_{i_j}$

Exmpl: $X = A\,B\,C\,B\,D\,A\,B$, $Z = B\,C\,D\,B$

$\Rightarrow$ Z can be obtained from X by removing elements from X. & keep order of remaining elements.

IF Z is subsequence of X & Y $\Rightarrow$ Z is common subsequ. of X & Y.

Exmpl: $X = A\,B\,C\,B\,D\,A\,B$

$Y = B\,D\,C\,A\,B\,A$

$Z = BCA$, $Z' = BCBA$, $Z'' = BDAB$ are common subsequ. of X & Y.

Aim: Find longest common subsequ. (LCS) of X & Y.

Brute FORCE is not a good idea, Since $X = x_1 \cdots x_m$ has 2^m subsequences!

For $X = x_1 \cdots x_m$, define the i-th prefix X_i of X as $X_i = x_1 \cdots x_i$, $1 \le i \le m$ & put $X_0 = \varepsilon$ "empty sequ."

Step 1: Characterization of LCS.

Theorem 4.2 [Opt. substructure of LCS]

Let $X = x_1 \dots x_m$, $Y = y_1 \dots y_n$ be two sequences.
& $Z = z_1 \dots z_k$ be some LCS of X & Y.

(1) $x_m = y_n \Rightarrow z_k = x_m = y_n$ & z_{k-1} is LCS of x_{m-1} & y_{n-1}

(2) $x_m \neq y_n$ & $z_k \neq x_m \Rightarrow Z$ is LCS of x_{m-1} & Y

(3) $x_m \neq y_n$ & $z_k \neq y_n \Rightarrow Z$ is LCS of X & y_{n-1}.

Illustration: (1)

$X = A\ A\ B\ A\ C\ D$
$Y = A\ B\ B\ C\ D$
LCS $Z = A\ B\ C\ D$
(Z_3)

$\Rightarrow z_3$ is LCS of $X_5 = X_{m-1}$, $y_4 = y_{n-1}$, $m = 6, n = 5$

$\Rightarrow$ can reduce problem to find LCS of x_{m-1} & y_{n-1}

(2)

$X = A\ A\ B\ A\ C\ A$
$Y = A\ B\ B\ C\ D$
LCS $Z = A\ B\ C$

! this is independent of letter y_n that is both cases may occur: $z_k = y_n$ or $z_k \neq y_n$

But since z_k never "matches" x_n, we can reduce the problem to find LCS of x_{m-1} & Y

(3) analog to (2).

proof: [Thm 4.2]

(1) $x_m = y_n \Rightarrow z_k = x_n$ [otherwise if $z_k \neq x_n$ we can append x_n to $\underbrace{z_1 \dots z_k}_{\text{already CS}}$ & get larger subsequence $\lightning$ since z is LCS.]

$\Rightarrow x_m = x_n = z_k$

Now prefix z_{k-1} is a common subseqn. of x_{m-1} & y_{n-1}

to show: z_{k-1} is $\underline{\underline{L}}$CS of x_{m-1} & y_{n-1}

But this is clear, since if there is a longer one,

say W then we could append $z_k + W$ to obtain

a seqn. which is longer than z $\lightning$ z is LCS

(2) Clearly if $z_k \neq x_m \Rightarrow z$ is common subseq. of x_{m-1} & y.

to show z is $\underline{\underline{L}}$CS of x_{m-1} & y.

Again, if there is a longer common subseqn. W of x_{m-1} & y

then W is also a common subseq. of x & y,

but longer than z $\lightning$ z is LCS

(3) Analog to (2)

□

Step 2: recursive solution of LCS

By Thm 4.2: $x_m = y_n \Rightarrow$ Find LCS of X_{m-1} & Y_{n-1} & append $x_m = y_n$ to this LCS.

$x_m \neq y_n \Rightarrow$ Find LCS of x_{m-1} & y or x & y_{n-1} whichever of these is longer is an LCS of x & y

Let c_{ij} = length (# of letters) of LCS of prefixes X_i & Y_j

NOTE $x_0 = \varepsilon$, $y_0 = \varepsilon \Rightarrow c_{ij} = 0$, if $i = 0$ or $j = 0$

Recursive Formula:

$$c_{ij} = \begin{cases} 0 & \text{, if } i=0 \text{ or } j=0 \\ c_{i-1,j-1} + 1 & \text{, } i,j>0 \text{ \& } x_i = y_j \\ \max\{c_{i,j-1}, c_{i-1,j}\} & \text{, } i,j>0 \text{ \& } x_i \neq y_j \end{cases}$$

Step 3: Computing length of LCS.

Based on recursive formula for $c[i,j]$
we get for free an exponential-time recursive alg.
But DP can be used to get it in polynomial time.

Let $X = x_1 \cdots x_m$ & $Y = y_1 \cdots y_n$

```
LCS(X,Y)
    Let b[1...m, 1...n]                 // b used to construct LCS via backtracking
        c[0...m, 0...n] be new arrays   // c stores length.

    FOR (i = 0...m) DO  c[i,0] = 0
    FOR (j = 0...n) DO  c[0,j] = 0

    FOR (i = 1...m) DO
        FOR (j = 1...n) DO
            IF (x_i = y_j)
                c[i,j] = c[i-1,j-1] + 1
                b[i,j] = "↖"
            ELSE IF (c[i-1,j] ≥ c[i,j-1])
                c[i,j] = c[i-1,j]
                b[i,j] = "↑"
            ELSE
                c[i,j] = c[i,j-1]
                b[i,j] = "←"

    RETURN c & b.
```

// c:

$\Rightarrow x_i = x_j$ is part of LCS ☺

Step 4: construct LCS:

```
PRINT-LCS(b, X, i, j)   // initial call: PRINT-LCS(b, X, m, n)
    IF (i = 0 or j = 0) RETURN.
    IF (b[i,j] = "↖")
        PRINT-LCS(b, X, i-1, j-1)
        print x_i
    ELSE IF (b[i,j] = "↑")
        PRINT-LCS(b, X, i-1, j)
    ELSE
        PRINT-LCS(b, X, i, j-1)
```

RUNTIME: $\Theta(m+n)$
since it decrements at most one of i & j in each call

Exmpl:

$X = ABCB$, $Y = BDC$

i	j	0	1	2	3
			B	D	C
0		0	0	0	0
1	A	0			
2	B	0			
3	C	0			
4	B	0			

now $i = 1..m$ / $j = 1..m$

$i = 1$, $j = 1,2,3$

$i = 2$, $j = 1,2,3$

$x_2 = y_1$ "↖"

$i = 3$, $j = 1,2,3$ $x_3 = x_4$ "↖"

$i = 4$, $j = 1,2,3$ $x_4 = y_1$ "↖"

i	j	0	1	2	3
			B	D	C
0		0	0	0	0
1	A	0	0	0	0
2	B	0	1	1	1
3	C	0	1	1	2
4	B	0	1	1	2

Backtracking:

start at $c[m,n]$

"order" ↖ , ↑ , ←

but there might be further opt. solutions!

$b[3,3] =$ ↖ → print 'C'

$b[2,1] =$ ↖ → print 'B'

$\Rightarrow Z = BC$

this value is length of LCS

Theorem 4.3: LCS(·) + PRINT-LCS(·) correctly returns length & LCS of given $X = x_1 \dots x_m$ & $Y = y_1 \dots y_n$ in $\Theta(mn)$ time.

proof: correctness by Thm 4.2

runtime dominated by 2 FOR-loops ($i = 1..m$, $j = 1..n$)

$\Rightarrow \Theta(mn)$

/13

"Non-exact" matching (Alignments & Co)

so far: mostly "exact" matching (except LCS)

Differences due to seq. error, natural variations
↓
insertion, del, additions —

→ need "approx. matching"

want distance between 2 strings.
to get an idea on "how similar" to strings are.

Hamming distance

Simplest idea = Hamming distance, def for 2 strings of same length.

for $S = s_1 \dots s_n$, $S' = s_1' \dots s_n'$ def $d_{Hamm}(S,S') = \sum_{i=1}^{n} \delta(s_i, s_i')$

where $\delta(s_i, s_i') = \begin{cases} 0, & s_i = s_i' \\ 1 & \text{else.} \end{cases}$

[$d_{Hamm}(S,S')$ = # mismatches]

Example: $S = ATCATGA$, $S' = ATGATCA$ $\Rightarrow d_{Hamm}(S,S') = 2$

problem, in general, this is not what is meant with similarity from a biolog. POV.

$S = ATATATATATA$, $S' = TATATATATAT$ } very similar in their 'characteristic'

but $d_{Hamm} = 9$

need something else.

LCS

S: GC

S_1: GCGC → LCS = 2

S: GC

S_1: GCGCGCGCGC → LSC = 2

Both S & S_1, S & S_2 have same LCS score,
although S & S_1 "are more sim. lar" than S to S_2

LCS is good for finding inexact "subpattern"
but not for comparing similarity between seqn.

(LCS does not take into account insertion/deletions)

Edit-Distance (Levenshtein-Distance) & Alignments

when genes get copied etc:

ATGCGT → AGCGT — T on pos. 2 removed [deletion]

ATGCGT → ATGGCGT — G on pos 4 added [insertion]

ATGCGT → TTGCGT — A on pos 1 replaced by T [substitution (mismatch)]

DEF edit-distance of s, s' $d_{Edit}(s,s')$ = min Nr of deletion, insertion, substitions to transform s into s'

Observation:

- s, s' don't need to be of same length
- $d_{Edit}(s,s') = d_{Edit}(s',s)$ (symmetric)
- $||s|-|s'|| \leq d_{Edit}(s,s') \leq d_{Hamm}(s,s')$
- edits don't need to be unique. $s = AT$, $s' = G$ (remove A, replace T by G; or remove T, replace A by G)

$s = TGCATAT \qquad s' = ATCCGAT$

TGCATAT → delete T at pos 7

TGCATA → delete A at pos 6

TGCAT → insert A before pos 1

ATGCAT → replace G by C

ATCCAT → insert G before A

ATCCGAT

$\Rightarrow d_{Edit}(s, s') \leq 5$

TGCATAT → add A before pos 1

ATGCATAT → delete T pos 6

ATGCAAT → replace A pos 5 by G

ATGCGAT → replace G pos 3 by C

ATCCGAT

$\Rightarrow d_{Edit}(s, s') \leq 4.$

But what is min dedit ???

↳ detour along alignments

DEF: Alignment $d(s s')$ of s, s', $s, s \in \Sigma^*$ (Σ alphabet)

is $2 \times n$ matrix, $n \geq \max\{|s|, |s'|\}$

such that

- each row consist of characters in Σ & '–' gap symbol
- deletion of gaps in each row yields s resp s'.
- in no column are 2 gaps

Exampl

$S = TGCATAT$

$S' = ATCCGAT$

$$\mathcal{A}(S,S') = \begin{pmatrix} -TGCATAT \\ ATCCG-AT \end{pmatrix}$$

insertion — match — replacement (mismatch) — deletion

DEF : For given alphabet Σ, let

$$\delta : (\Sigma \cup \{-\}) \times (\Sigma \cup \{-\}) \longrightarrow \mathbb{R}$$

be a cost-function

[common case: $\delta(x,y) = \begin{cases} 1 \text{ if } x \neq y \\ 0, \text{ else} \end{cases}$ (unit score)]

$$\mathcal{A}(S,S') = \begin{pmatrix} a_1 \dots a_\ell \\ b_1 \dots b_\ell \end{pmatrix} \Rightarrow \delta(\mathcal{A}) = \sum_{j=1}^{\ell} \delta(a_j, b_j)$$

Exampl.

$$\begin{pmatrix} -TGCATAT \\ ATCCG-AT \end{pmatrix}$$

unit score: $1\,0\,1\,0\,1\,1\,0\,0 = 4$

Aim: Find alignment $\mathcal{A}$ that minimizes $\delta(\mathcal{A})$ (= optimal Alignment)

Lemma 1 $\min d_{Edit}(s,s') = \min_{alt} \delta(alt(s,s'))$ for δ = unit costs.

proof: alt yields edit-operat. & vice versa

$\min \delta_{edit}(s,s') \leq \min_{alt} \delta(alt(s,s')) \Rightarrow "="$. □

In general, costs diffs from unit costs.

e.g $\delta(x,x) = -2 \quad \forall x \in \{A,C,T,G\}$

$\delta(x,-) = \delta(-,x) = 1$

$\delta(A,C) = 5$

⋮

DEF: let s,t be strings, δ be cost-fct.

Then $D(i,j)$ = min costs of alignments of $s[1..i]$ & $t[1..j]$

[that is $D(|s|,|s'|) = \delta(alt(s,s'))$]

Lemma 2 Let $S = s_1 \dots s_k$, $T = t_1 \dots t_\ell$ strings

Then, for all $i, j \geq 1$ it holds that

$$D(i,j) = \min \begin{cases} D(i-1,j) + \delta(s_i, -) \\ D(i, j-1) + \delta(-, t_j) \\ D(i-1, j-1) + \delta(s_i, t_j) \end{cases}$$

where

$$D(0,0) = 0$$
$$D(0,j) = D(0, j-1) + \delta(-, t_j)$$
$$D(i,0) = D(i-1, 0) + \delta(s_i, -)$$

$D(0,0) = 0$ $\hat{=}$ align <u>empty string ε</u> with ε

$D(i,0)$ $\hat{=}$ align $S[1..i]$ with ε
$\hat{=}$ delete all characters $s_1 \dots s_j$ to get ε
$\hat{=}$ $\sum_{r=1}^{i} \delta(s_r, -)$

$D(0,j)$ $\hat{=}$ align ε with $T[1..j]$
$\hat{=}$ added $t_1 \dots t_j$ to ε to get $T[1..j]$
$\hat{=}$ $\sum_{r=1}^{j} \delta(-, t_r)$

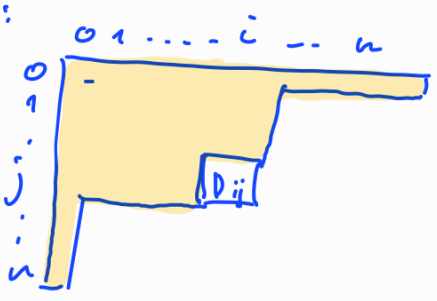
in case of unit costs: $D(0,0) = 0$
& $D(i,0) = i$ for $i \geq 1$
& $D(0,j) = j$ for $j \geq 1$

proof of Lemma 2

Induction. , $D(0,0)$
$D(0,i)$ $i \geq 1$
$D(j,0)$ $j \geq 1$ are correct.

⇒ when computing $D(i,j)$ we assume that all previous values have been correctly computed:

D:



Consider $D(i,j)$.

Let A_{ij} be opt alignment for $s[1..i]$ & $T[1..j]$

⇒ 3 cases how A_{ij} "ends"

(a) $\left(\begin{array}{|c|}\hline \cdots \\ \hline A \\ \hline \cdots \\ \hline \end{array}\begin{array}{c} s_i \\ - \end{array}\right)$ (b) $\left(\begin{array}{|c|}\hline \cdots \\ \hline B \\ \hline \cdots \\ \hline \end{array}\begin{array}{c} - \\ t_j \end{array}\right)$ (c) $\left(\begin{array}{|c|}\hline \cdots \\ \hline C \\ \hline \cdots \\ \hline \end{array}\begin{array}{c} s_i \\ t_j \end{array}\right)$

Suppose case(a)

since A_{ij} opt alignment for $s[1..i]$ & $T[1..j]$

⇒ A opt alignment for $s[1..i-1]$ & $T[1..j]$

(if not ⇒ ∃ A'' better alignment for $s[1..i-1]$ & $T[1..j]$
now add column $\left(\begin{smallmatrix} s_i \\ - \end{smallmatrix}\right)$ to A''
to get better Alignment than A_{ij} ↯)

⇒ $\delta(A) \overset{\text{ind hyp}}{=} D(i-1,j)$

⇒ $\delta(A_{ij}) \overset{\text{Def}}{=} D(i,j) \overset{\text{Alg}}{=} D(i-1,j) + \delta(s_i,-)$
is correctly computed

analog for case (b) & (c)

(b) B opt alignment for $S[1..i]$ & $T[1..j-1]$

(c) C opt alignment for $S[1..i-1]$ & $T[1..j-1]$.

& $\delta(t_{ij})$ correctly computed.

under the assumption that we have case a) b) c)

However at least one case must occur & no further case can occur.

$$\Rightarrow \underbrace{\delta(t_{ij})}_{\text{"def } D(i,j)} = \min \begin{cases} D(i-1,j) + \delta(s_i, -) \\ D(i, j-1) + \delta(-, t_j) \\ D(i-1, j-1) + \delta(s_i, t_j) \end{cases}$$

$\Rightarrow \delta(\text{Lcf}(S,T)) = D(|S|, |T|)$ correctly determined ∎

```
globalAlign(S, T, δ)        // S = s_1 ... s_k,  t = t_1 ... t_ℓ

    Init:   D(0,0) = 0
            FOR( i = 1 ... k )
                D(i,0) = D(i-1,0) + δ(s_i, -)
            FOR( j = 1 ... ℓ )
                D(0,j) = D(0,j-1) + δ(-, t_j)

    FOR (i = 1 ... k)
        FOR( j = 1 ... ℓ )
            D(ij) = min { D(i-1,j) + δ(s_i, -)
                        { D(i,j-1) + δ(-, t_j)
                        { D(i-1,j-1) + δ(s_i, t_j)

    Return D
```

By the latter arguments & Lemma 2,

globalAlign correctly computes min-edit cost
(opt. alignment score)

Runtime: $O(|s|\,|t|)$

Example, δ = unit costs, s = TbCA, t = ATC

		0	1	2	3	4
	D	ε	T	b	C	A
0	ε	0	1	2	3	4
1	A	1	1	2	3	3
2	T	2	1	2	3	4
3	C	3	2	2	2	(3)

(3) = D(|s|, |t|) = min-edit costs.

So far only know value of min-edit costs,
But not the alignment.

⇒ TRACEBACK (can be computed when computing D)

		0	1	2	3	4
	D	ε	T	b	C	A
0	ε	0	1	2	3	4
1	A	1	1	2	3	3
2	T					
3	C					

want to compute D(1,4)

$D(1,4) = \min(D(1,3)+1,\ D(0,4)+1,\ D(0,3)+0)$

$= \min(4,\ 5,\ 3)$

in this case "unique" path that is D(1,4) can only be computed by using entry D(0,3)

& this info can be stored when computing D(1,4)

		0	1	2	3	4
	D	ε	T	b	C	A
0	ε	0	1	2	3	4
1	A	1	1	2	3	3
2	T	2	1	2		
3	C					

$D(2,2) = \min(D(2,1)+1,\ D(1,2)+1,\ D(1,1)+1)$

$= \min(2,\ 3,\ 2)$

⇒ D(2,2) can be computed using "coming from" D(2,1) or D(1,1)

		0	1	2	3	4
	D	Σ	T	G	C	A
0	Σ	0	1	2	3	4
1	A	1	1	2	3	3
2	T	2	1	2	3	4
3	C	3	2	2	2	3

Now follow one "path" from $D(|S|,|T|)$ back to $D(0,0)$ to construct alignment as follows:

IF $(i-1,j) \rightarrow (i,j)$ $\Rightarrow D(i-1,j) + \delta(s_i, -)$

$\Rightarrow \begin{pmatrix} .. & s_i & ... \\ & - & \end{pmatrix}$

IF $(i-1,j-1) \rightarrow (i,j)$ $\Rightarrow D(i-1,j-1) + \delta(s_i, t_j)$

$\Rightarrow \begin{pmatrix} & s_i & \\ & t_j & \end{pmatrix}$

IF $(i,j-1) \rightarrow (i,j)$

$\Rightarrow D(i,j-1) + \delta(-, t_j)$

$\Rightarrow \begin{pmatrix} & - & \\ & t_j & \end{pmatrix}$

		0	1	2	3	4
	D	Σ	T	G	C	A
0	Σ	0	1	2	3	4
1	A	1	1	2	3	3
2	T	2	1	2	3	4
3	C	3	2	2	2	3

Have 2 possible paths

$\Rightarrow$ 2 possible opt. alignments.

$\begin{pmatrix} T & G & C & A \\ A & T & C & - \end{pmatrix}$

$\begin{pmatrix} - & T & G & C & A \\ A & T & - & C & - \end{pmatrix}$

Exampl shows, since no penalty for gaps, we may have a lot of them.

Alg globalAlign also better know as

Needleman-Wunsch-algorithm

(1970)

(also globalAlign slightly deviates, since original Needleman-Wunsch Alg had runtime $O(|S|^2|T|^2)$

→ $O(|S||T|)$ is runtime of Wagner/Fischer 1974)

GAPS: ACC GTCTGCT

A-C--C-G-T

ACC GTCTGCT

ACC GT-----

in evolution, "consecutive segm." of gaps easier to realize than a lot of single gaps.

& also to avoid "unnecessary" gaps as in

$\begin{pmatrix} TGCA \\ ATC- \end{pmatrix}$

$\begin{pmatrix} -TGCA \\ AT-C- \end{pmatrix}$

Def: gap penalty function $g: \mathbb{N} \to \mathbb{R}$

s.t. $g(k+n) \leq g(k) + g(n)$ ("sub-additiv")

$g(i)$ = penalty for having i consecutive gaps.

$g(k+n) \leq g(k) + g(n)$ ensures that it is better to have $k+n$ "consecutive" gaps than one somewhere of length k & — " — n.

Exempl $g(i) = i + 1$, unit-cost δ.

```
A C C G T C T G C T        A C C G T C T G C T
A - C - - C - G - T        A C C G T - - - - -
0+2+0+3+0+2+0+2+0 = 9      0   + g(5) = 0 + 6 = 6
```

Alignment with gap penalties

(as above but modified "D")

$$D(0,0) = 0, \quad D(0,k) = D(k,0) = g(k), \quad k \geq 1$$

$$D(i,j) = \min \begin{cases} D(i-1, j-1) + \delta(u_i, v_j) \\ \min_{1 \leq k \leq i} D(i-k, j) + g(k) \\ \min_{1 \leq k \leq j} D(i, j-k) + g(k) \end{cases}$$

Problem: $O(|S||T|(|S|+|T|))$ "Cubic"

Solution: affine gap costs

$$g(k) = \alpha + \beta \cdot k$$

α = cost for "opening" gaps

β = costs for extending gaps.

GOTOH (1982),
Altschul & Erickson (1986)]

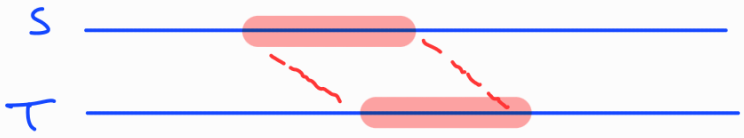
↳ $O(|S||T|)$ - time Alg

So-far, what we have done is <u>global</u> Alignment

that is,

Compare entire S with entire T

<u>local</u>: Find most similar pair of substrings of S & T



in practice, matches get pos. weight (high similarity)

deletion/insertion neg-weight

in this case we call δ scoring fct.

Exmpl

δ	-	A	C	G	T
-		-6	-6	-6	-6
A	-6	2	-4	-4	-4
C	-6	-4	2	-4	-4
G	-6	-4	-4	2	-4
T	-6	-4	-4	-4	2

gaps

matches

mismatches

XX

l_D (local D):

$$l\_D(0,:) = l\_D(:,0) = 0 \quad \forall\, i \geq 0$$

[but substring of S & ε (resp ε & T) is the empty string ε = "0"]

$$l\_D(i,j) = \max \begin{cases} l\_D(i-1,j) + \delta(s_i, -) \\ l\_D(i,j-1) + \delta(-, t_j) \\ l\_D(i-1,j-1) + \delta(s_i, t_j) \\ 0 \end{cases}$$

with δ = scoring function.

Example: $S = GCCG$, $S' = ACCA$, δ as above

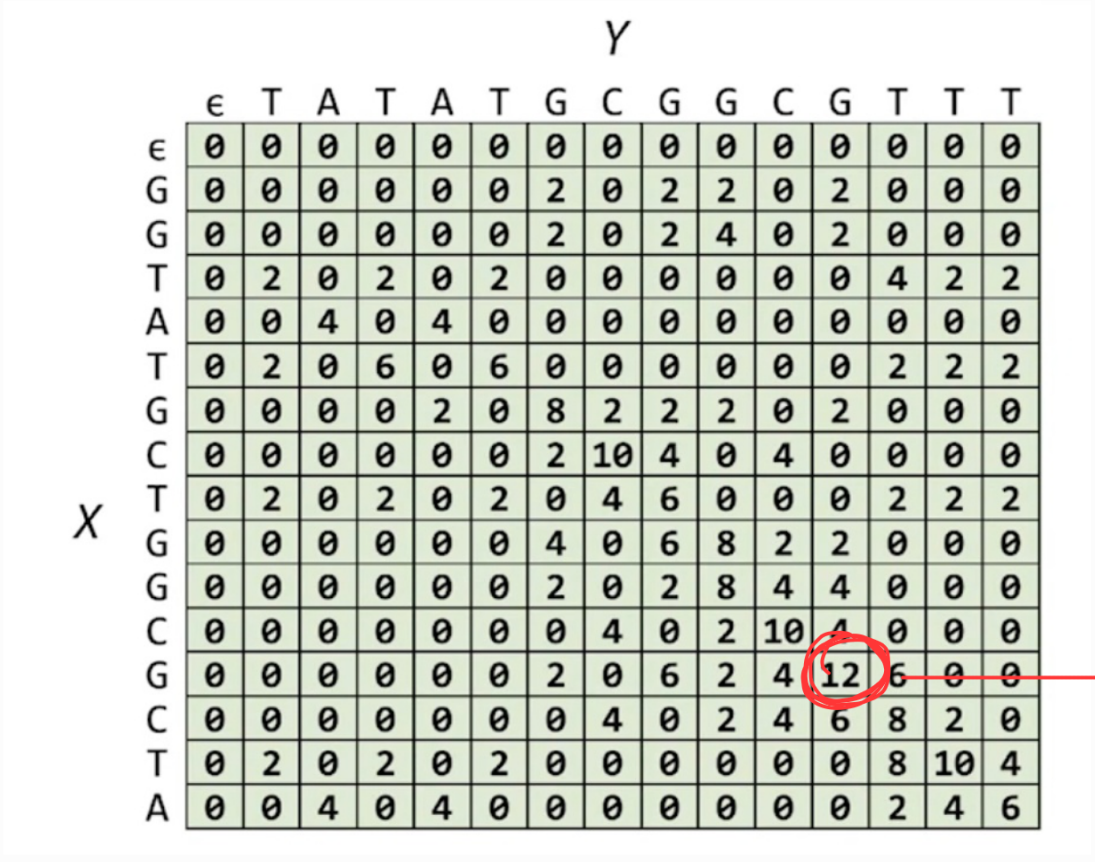
lD	ε	G	C	C	G
ε	0	0	0	0	0
A	0	0	0	0	0
C	0	0	2	0	0
C	0	0	2	4	0
A	0	0	0	0	0

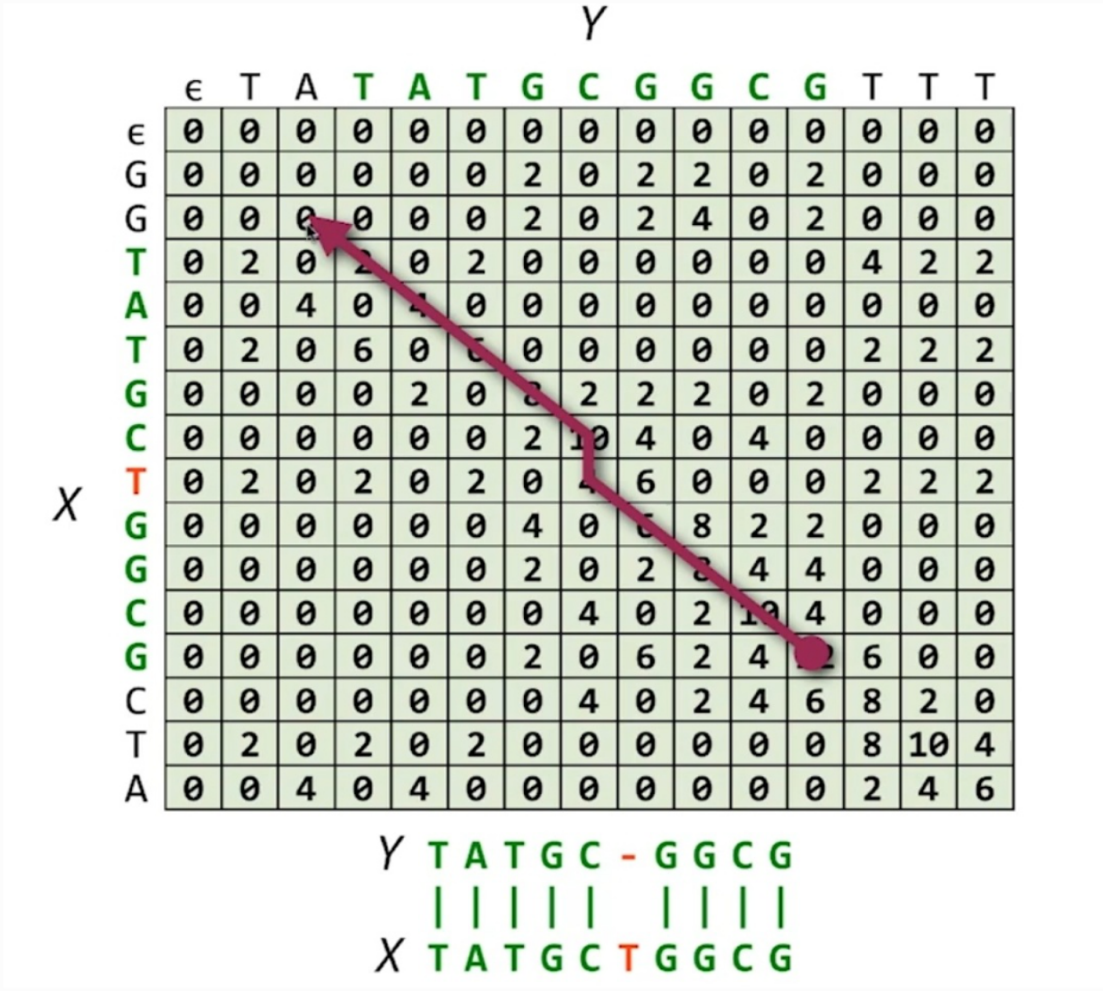
max + traceback to first '0'

⇒ get CC as best local alignm.

(this is the basic idea that together with gap-penalty function is known as Smith-Waterman-Alg)

Another larger Example:





Protein comparison & SCORING Functions

(max instead of min in Alg above)

$$\text{LCS} \hat{=} \quad \sigma(x,y) = \begin{cases} 1 & x = y \text{ (match)} \\ -\infty & \text{mismatch (not allowed)} \\ 0 & \text{insert, deletion,} \\ & \text{i.e. } x = - \text{ or } y = - \end{cases}$$

simple idea:

eg. In Humans: A ↔ G, C ↔ T substitution ~2 times more frequent than other

A, G ∈ Purines

C, T ∈ Pyrimidins

& deletion/insertion less frequent than substitution

	-	A	C	G	T
-		-6	-6	-6	-6
A		2	-4	-2	-4
C			2	-4	-2
G	(sym.)			2	-4
T					2

Scores.

(substitution ~1 in 1000

Small gaps: ~1 in 3000

(take two human genomes & compare)

Often protein-seqs. instead of DNA-seqs. are compared.

Why?

- AAG / AAA → ≠0 Alignm. score

  But both AAG & AAA encode Lysine same aminoacid.

  "Silent mutations"

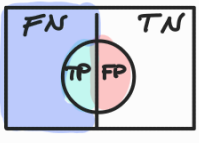
  AAG / TAG → ≠ Alj score but possibly same score (similar) as AAG, AAA

  But AAG → Lysine
  TAG → STOP → yields possibly fatal protein

  comparing protein sequences "more robust"

- due to genetic code, very similar proteins might be encoded by pair of DNA sequences that share only limited similarity

- 20 Aa instead 4 Bases ⇒ higher sensitivity & shorter sequences (runtime)



$$\text{sensitivity} = \frac{TP}{FN+TP} = \text{true pos. rate}$$

How to obtain "good" scoring ?

As in example "simple idea",

Observation: Asn, Asp, Glu, Ser most "mutable" amino acid

Cys & Trp are the "least" mutable

eg. Ser → Phe is ~3 times higher than Trp → Phe

Based on this knowledge one can start to design scoring matrices.

Scoring matrix δ

$\delta(i,j)$ usually reflects how often amino acid i substituted amino acid j

↓

if we have large sets of alignments

$\delta(i,j)$ simply counts how many times i is replaced by j.

Problem: to get alignments, we need already some scoring function.

But for similar sequences, unit-score can be used first & then δ can be established & used to construct less "obvious" alignments.

highly (> 90%)

Multiple Sequ. alignments (MSA)

So-far 2 sequences only.

Formal

set of k sequences $S = \{s_1 \dots s_k\}$, $s_i \in \Sigma^*$

multiple sequ. alignment $A(s_1 .. s_k)$ is $k \times n$ matrix
$(n \geq \max\{|s_1| .. |s_k|\}$

it:

- each row consist of characters in Σ & '-' gap symbol
- deletion of gaps in each row i yields s_i
- in no column all rows are filled with gaps.

s_1 A C G - - G A G A
s_2 - C G T T G A C A
s_3 A C - T - G A - A
s_4 C C G T T C A C -

possible score: Sum-of-Pair (SP) score:

$$SP(S_1 .. S_k) = \sum_{1 \le i < j \le k} \sigma(S_i S_j) \quad ,$$

where $\sigma(S_i, S_j)$ = "classical" alignm. score between S_i & S_j

```
S1   A C G - - 6 A 6 A
S2   - C G T T 6 A C A
S3   A C - T - 6 A - A
S4   C C G T T C A C -
```

match = 2
mismatch = -2
gap open = -2
gap extension = -1

$\Rightarrow$ $\sigma(S_1 S_2)$:

```
S1   A C G - - 6 A 6 A
S2   - C G T T 6 A C A
     -2 2 2 -2-1 2 2-2 2
```

= 3

$\sigma(S_1 S_3) = 4$
$\sigma(S_1 S_4) = -5$
$\sigma(S_2 S_3) = 2$
$\sigma(S_2 S_4) = 6$
$\sigma(S_3 S_4) = -6$

$\Rightarrow$ $SP(S_1 - S_k) = 3 + 4 - 5 + 2 + 6 - 6 = 4$

Problem: finding and a multiple alignment with opt score is NP-hard.

Plenty of several methods to compute optimal MSA exists.

- exponential runtime algorithm
  (in part will runtime improvements via e.g. simulate annealing, heardspace-pruning (Grand Eband))
- approx. alg. (Simplified: polytime MSA of solution (distance) is at most twice the opt one)
- heuristics → progressive approaches (eg. ClustalW)
  → iterative — " — (eg. MUSCLE)
  (most commonly used)
- probabilistic methods
  (assumption on certain evol. models)
  (eg markov models)
  profile HMM.

Classical Alg based or concerned on w,l

Alignments:

- Blast
- Clustal
- MUSCLE.

BLAST = Basic Local Alignment Search Tool

Umbrella term for a collection of the world's most widely used programs for analyzing biological sequence data

- BLAST is used to compare experimentally determined DNA or protein sequences with sequences already existing in a database.
- Basic idea: BLAST divides query sequences into short strings and initially only looks for (exact) matches of those strings in database strings.
  This is afterwards extended to get the entire alignment.
- very fast local alignment heuristic, but no optimality guarantee
- output: series of local alignments, i.e. comparisons of pieces of the searched sequence with similar pieces from the database. In addition, BLAST indicates how significant of the hits that have been found.

Databases e.g. for nucleotide sequences (Genbank of NCBI, EMBL, . . .) or protein databases (SwissProt, RefSeq, Pfam, . . .).

BLAST homepage: `blast.ncbi.nlm.nih.gov`
Tutorial: `digitalworldbiology.com/BLAST`

BLAST"Types"

type	query	target
blastn	nucleotide	nucleotide
blastp	protein	protein
blastx	nucleotide (transl)	protein
tblastn	protein	nucleotide (transl)
tblastx	nucleotide (transl)	nucleotide (transl)

Example

`https://blast.ncbi.nlm.nih.gov/Blast.cgi` → nucleotide blast → copy&paste → press button BLAST

>Sequence_experimental
GACATTACGGCGACCCAGTCTCCCCCGGTGTTGTCAGTGGGACTGGGCC
AGACCGCAACCATCACTTGTACGGCCAGTCAAAGCATCTACAGTAACCT
TGCTTGGTACCAGCAGAGAGAAGGACAGAAGCCCTCTCTCCTGATCTAT
GCTGCGACAACGCGATACGAAGGAGTCTCCGAGCGATTCAGCGGCAGTG
GATCAGGGACCAGTTTCACCCTGACAATCAGCAACGTTCAGAATGAGGA
TGTCGCTGACTATTACTGTCAGATCGCATATTCGATCTACTCCGGTTCC
GTTGTTTTCGGTGAAGGAACCAAGCTCAGACTGAGCCGT

mRNA for some (specified) protein of a nurse shark.

Clustal

Clustal is a series of computer programs used in bioinformatics for multiple sequence alignment.

Brief History:

- Clustal (1981, first version)
- CLustalW (1994, great improvements)
- ClustalX (1997, first time with GUI)
- ClustalΩ (latest standard version, 2011)

Clustal

Basic idea explained on ClustalW (3 steps for input ζ= set $\{S_1, \ldots, S_k\}$ of sequences):

W1 Compute for all pairs $S_i, S_j \in \zeta$ a pairwise alignment $\Longrightarrow$ pairwise distances $D(S_i, S_j)$

W2 Use distance matrix D to compute phylogenetic tree T (via NeighborJoining-method)

W3 Use T to carry out a multiple alignment

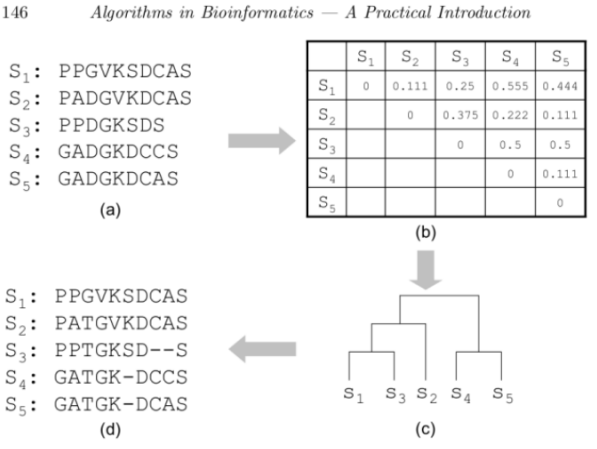


FIGURE 6.6: The three steps of ClustalW (a progressive alignment methods). Five input sequences are given in (a). Step 1 computes the pairwise distance scores for these five sequences (see (b)). Then, Step 2 generates the guide tree such that similar sequences are grouped together first (see (c)). Step 3 aligns the sequences one by one according to the branching order of the guide tree, yielding the multiple alignment of all input sequences (see (d)).

Clustal W: 3 steps W1, W2, W3

(W1) $\forall$ pairs S_i, S_j: compute optimal global alignment.

$\Rightarrow$ x = non gap pos. ("$\bar{c}$, $\bar{c}$")
y = #matches

$$d(S_i S_j) = 1 - \frac{y}{x}$$

PPGVKSDCAS
PPDGKSD--S opt alignment

$x = 8$, $y = 6$ $\rightsquigarrow$ $d(S_i, S_j) = 1 - \frac{6}{8} = 0{,}25$.

$\longrightarrow$ this gives distance matrix D

(W2) use D to compute phyl. tree (Later in course)

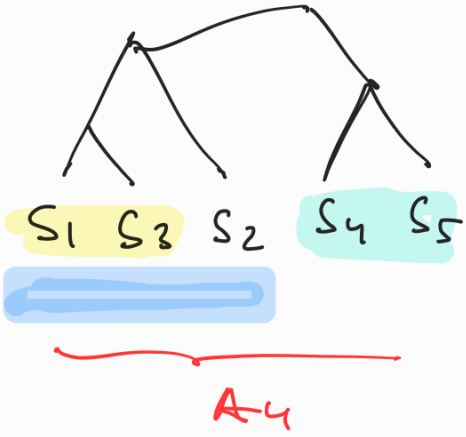
(W3)

align $(S_1, S_3) \rightsquigarrow A_1$

align $(S_4, S_5) \rightsquigarrow A_2$

align $(A_1, S_2) \rightsquigarrow A_3$

align $(A_3, A_2) \longrightarrow A_4$.



RUNTIME $O(k^2n^2 + k^3)$ $n = \max_{i=1..k} \{|S_i|\}$

MUSCLE (multiple seqn. comparison by log expectation

MUSCLE

MUltiple Sequence Comparison by Log-Expectation (2004)

computer software used in bioinformatics for multiple sequence alignment.

Online available via `https://www.ebi.ac.uk/Tools/msa/muscle/`

Basic idea for input ζ= set $\{S_1, \ldots, S_k\}$ of sequences (2nd and 3rd steps similar to ClustalW):

1. Compute k-mer distances
   =(dis)similarities $D(S_i, S_j)$ between the sets of k-mers for all pairs $S_i, S_j \in \zeta$
   Much(!) faster than [W1] in Clustal

W2 Use distance matrix D to compute phylogenetic tree T (via UPGMA-method)

W3 Use T to carry out a multiple alignment

4. Several re-iteration and refinement steps follow

for $S = \{S_1 \dots S_n\}$ set of sequ.

compute k-mer "distances"

(not pairw. alignment)

k-mer = contiguous substring of length k.

⇒ distance matrix.

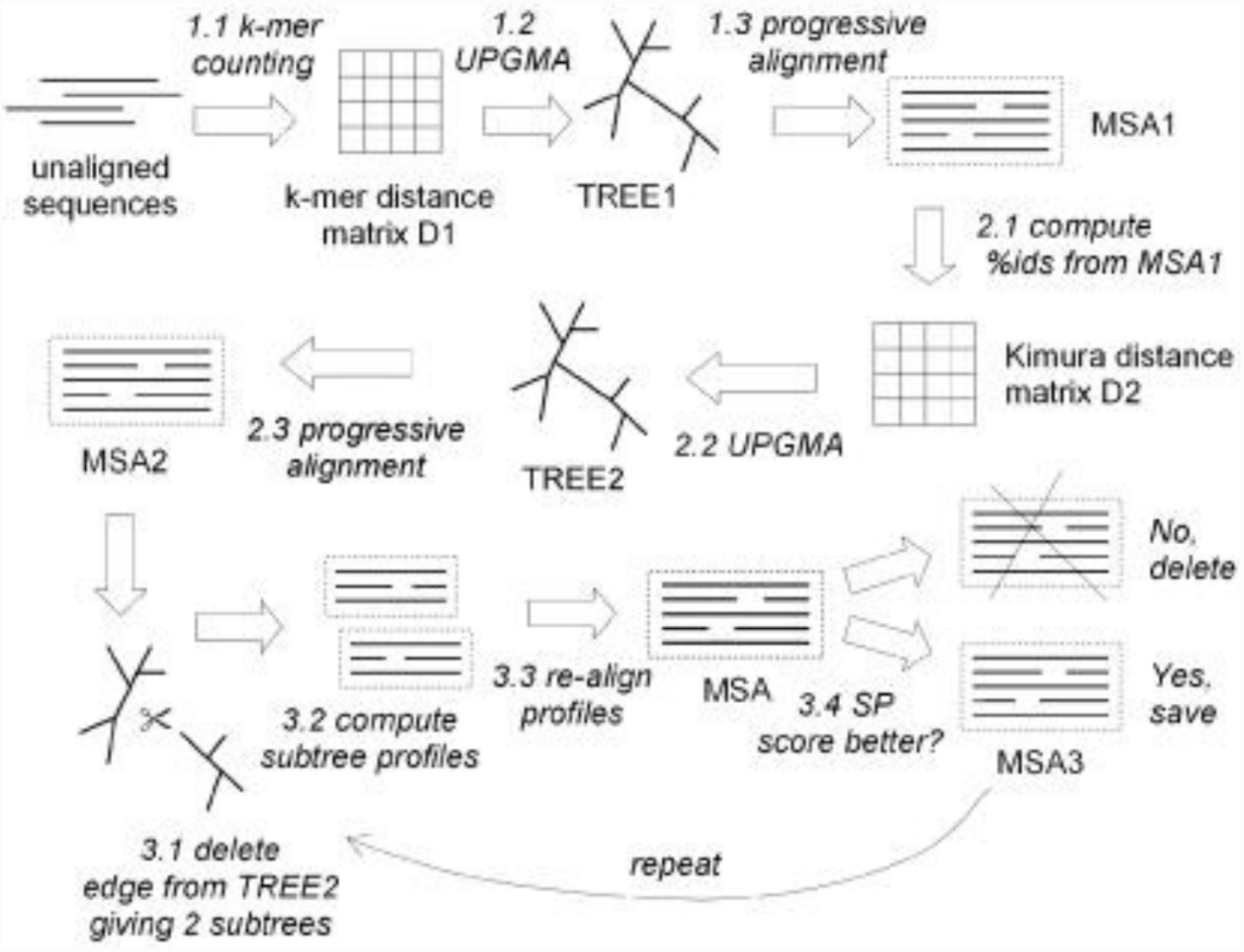
↓ W2/W3 as in ClustalW

this step is considerably faster than W1 in ClustalW

After this step several "recomputing" & "refinement" steps follow.

[ClustalW uses neighbor-joining ($O(n^3)$)

Muscle UPGMA ($O(n^2)$) to compute trees]



In general more accurate & faster than
ClustalW.