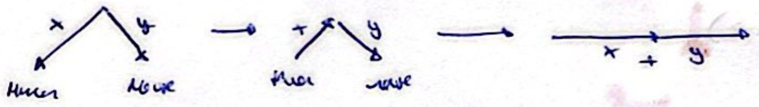
# LECTURE 2. Dynamic Programming & Sequence Aligment

Every organism is thought to come from a common ancestor.
Genome-wide alignments { - highly preserved areas ⟶ high functional

Evolution preserves <u>functional elements</u>

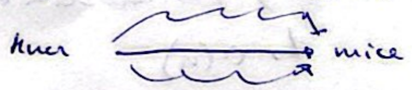
Genomes change over time: mutation, deletion... S. alignment allows us to find these changes.

- Insertion, deletion, mutation ⟶ symmetric operations (i.e., reversibility).



! Exception: CpG dinucleotides aren't simmetric

- Optimality criterion: min number / cost.



- Design algorithm (tradeoff)

Example: Formulation 1: Largest Common Substring

S1: A CGT C A T C A
S2: T A G T G T G A

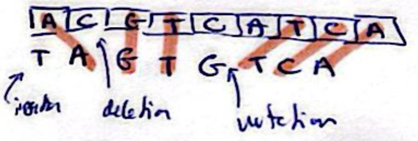
$\xrightarrow{\text{offset: -2}}$

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

. Algorithm slow
↳ make it jump if mismatch → faster

Formulation 2 Longest Common Subsequence { given $X = (x_1, \dots, x_m)$, $Z = (z_1, \dots, z_k)$ subsequence of $X$ if $i_1 < i_2 < i_3 \dots < i_k$

Allow gaps



⇒ ~~less~~ len(LCSS) = 6.
uniform scoring function (= weights)

Formulation 3 — S. Align.
- Allow gaps (fixed penalty)
  insertion - deletion
- Varying penalties for edit
  transitions (pyrimidine ⟶ pyrimidine, purine ⟶ purine) ↓ cost
  transversion (purine ⟷ pyrimidine) ↑ cost

| | A | G | T | C |
|---|---|---|---|---|
| A | +1 | -1/2 | -1 | -1 |
| G | -1/2 | +1 | -1 | -1 |
| T | -1 | -1 | +1 | -1/2 |
| C | -1 | -1 | -1/2 | +1 |

* Account for ~~P~~ P(B)

Formulation 4 — Varying gap penalty
- Linear
- Affine penalty { 1st nucleotide → p ↑ ; : b (don't care about length of mismatch)
- General → length gap → cost
- Frame aware { multiple of 3 ; ↳ protein coding regions → less disruptive
- Seek duplications, rearrangements.

How many alignments can be there?

Longest non-boring alignment * (NBA — alignment w gaps always paired up w nucleotide)

$n = len(s1)$
$m = len(s2)$ } $l(lcss) = n+m$

$2^n$ order problem because @ each position there could be a gap.

↳ we need a polynomial algorithm to find best alignment amongst an exp. n° of alignments ⟶ $\boxed{\text{Dynamic Programming}}$

Intro to DP

Computing Fibonacci seq ⟶ $O(2^n)$ { trees nesting inside each other.

↑ $T(n) = T(n-1) + T(n-2) \ldots$

```
def fib(n):
    if n==1 or n==2: return 1
    return fib(n-1) + fib(n-2)

def fib(n)
    fib_t[1] = 1
    fib_t[2] = 1
    for i in range (3, n+1):
      fib_t[i] = fib_t[i-1] + fib_t[i-2]
    return fib_t[n]
```

Only fills one line of the tree.
↳ $O(n)$

To systematically use a bottom-up approach is successful

Overlapping problems → limited n°

Typically for optimization problems
↳ traceback → optimal path
↳ if dependencies between subproblems → no DP

In practice
Setting up:
1. Find matrix' parametrization (# dimensions, variables)
2. Make sure subproblems are finite
3. Transversal order (bottom up)
4. Recursion formula
5. Remember choices

Start:
1. Fill results, find op. score
2- Traceback

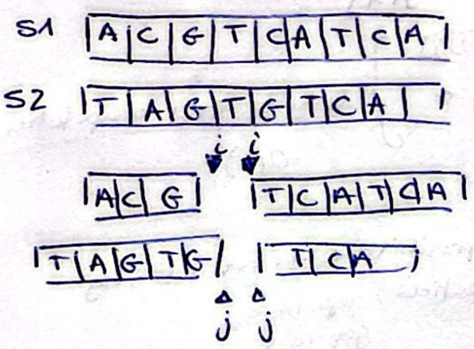
# Needleman - Wunch algoritm

**Score is additive, smaller to larger:**

→ for a given aligned pair (i,j) the best alignment is:

$$\left.\begin{array}{l} \text{Best of } S1[1\ldots i]\ \ S2[1\ldots j] \\ +\ \text{Best of } S1[i\ldots n]\ \ S2[j\ldots m] \end{array}\right\}$$ compute best alignment recursively!

Proof: cut & paste argument



⟹ Allows for a single recursion (top-left — bottom-right) instead of two (middle-to-outside top down)

**Compute score recursively:**

Solution #1: Memoization

- Create a dictionary indexed by aligned seqs when you encounter a new pair of seq
  - if it's in dict: look up the solution
  - if not, compute and add to the dic.
- Ensures no duplication of work!

  #Top down approach.

Solution #2: Dynamic programming
- Create a big table (i,j)

  Fill it

  Explores entire space.

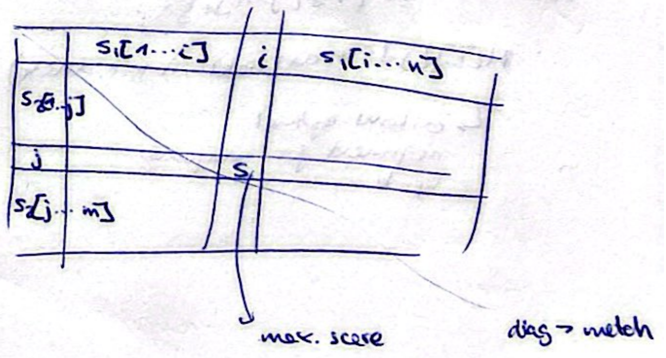
  No duplicated work.

  #Bottom up approach

**The optimal prefix alt score ⟷ matrix entry**

Every optimal algnt. → matrix path

↳ ~~best~~ algn ⟹ best path



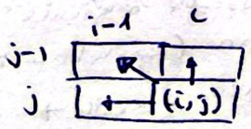
## DP Approach

- Compute all alignment scores from bottom-up:
  - Define $M[i,j]$ prefix alignment score of $S_1[1...i]$ and $S_2[1..j]$
  - Fill up table recursively from smaller to bigger alignments.

- Express alignment of $S_1[1... i+1]$ and $S_2[1...j+1] \rightarrow M[i+1, j+1]$
  - One of three possibilities: (1) extend alignment from $M[i,j]$
    (2) extend from $M[i-1,j]$ or
    (3) extend from $M[i,j-1]$

$\Big\}$ Only a local computation $O(1)$!

$$M(i,j) = \max \begin{cases} M(i-1,j) - gap \\ M(i-1,j-1) + score \\ M(i,j-1) - gap \end{cases}$$



Only 3 possibilities for extending by one nucleotide: gap in $S_1$
gap in $S_2$
a (mis) match

### Initialization

Create matrix of sizes $(m+1) \times (n+1)$
$len(S_1) = m$
$len(S_2) = n$

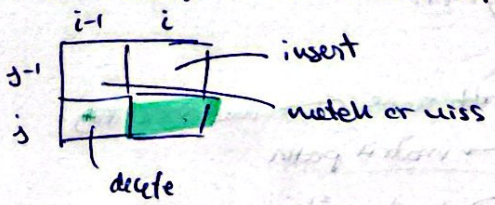
Each $M[i][j]$ represents the optimal alignment score for subsequences $S_1[1 \cdots i]$ & $S_2[1 ... j]$.

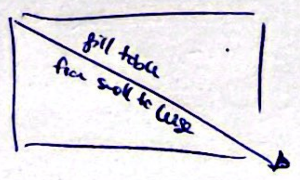
- $M[0][0]$ ~~gap~~ = 0
- $M[0][j]$ = gap-penalty * j
  1ª ~~fila~~ fila
- $M[i][0]$ = gap-penalty * i
  1ª ~~fila~~ column

### Fill Matrix

(1) Match or mismatch
~~match~~ match = M[i-1][j-1] + (matchscore if $s_1$[i-1] == $s_2$[j-1] else mismatch_penalty)
(2) delete = M[i-1][j] + gap
(3) insert = M[i][j-1] + gap

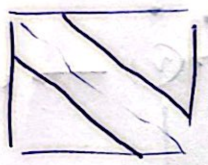
M[i][j] = max(match, insert, delete)
↳ contains optimal alignment for subsequences up to





## Advanced topics

We can realize that the path tends to be diagonal. Therefore, we can band the decisions by a diagonal width.



k(N)
↳ width

**Init**

F(i,0), F(0,j) undefined for i, j > k

**It.**

```
For i = 1 . . . M
    For j = max (1, i−k) . . . min (N, i+k)
```

Can we do better than $O(n^2)$?

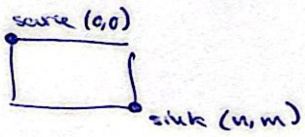
→ Probably not.

- - - - —

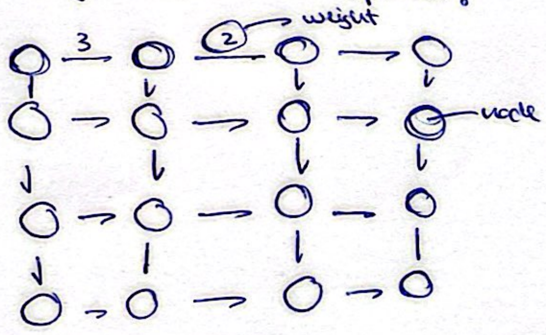
Manhattan Tourist Problem

Goal: to find the longest path (i.e. max (nº attractions or weights))

Vertices:
↳ source: northwesternmost point
sink: southeasternmost point



• Greedy solution is not optimal!



Approach
• Rather than (0,0) → (n,m) directly; (0,0) → (i,j) [arbitrary].
Best way from source to anywhere?

Find $s_{0,j}$ (for $0 \le j \le m$) is easy & $s_{i,0}$ ($0 \le i \le n$)
columns

$$\begin{tikzcd}
\boxed{0} \arrow[r, "3"] \arrow[d, "1"] & \boxed{3} \arrow[r, "2"] & \boxed{5} \arrow[r, "4"] & \boxed{9} \arrow[r, "0"] & \boxed{9} \\
\boxed{1} \arrow[d, "4"] \\
\boxed{5} \arrow[d, "4"] \\
\boxed{9} \arrow[d, "5"] \\
\boxed{14}
\end{tikzcd}$$

• Now we compute $s_{1,1}$
↳ $\overset{(0,1)}{\downarrow 0}$ } $s_{0,1}$ + weight of edge = $\boxed{3}$
$\underbrace{(1,0) \xrightarrow{3} (1,1)}_{s_{1,0} + \text{weigth} = 1+3 = \boxed{4}}$

• since goal is max. length: max $\begin{cases} s_{0,1} + \text{edge} = 3 \\ s_{1,0} + \text{edge} = 4 \end{cases}$ = 4
↓
Score for node (1,1)

## DAGs

The Manhattan problem can be solved as a no directed cycles DAG to consider non-perfect grids.

$$\begin{tikzcd}
u_1 \arrow[dr] & u_2 \arrow[d] & \\
u_3 \arrow[r] & v \arrow[r] \arrow[d] & w_2 \\
 & w_1 &
\end{tikzcd}$$

$G = (V, E)$

$E(u, v)$

- The nº of edges entering a vertex → indegree of $v$
- The nº of edges leaving a vertex → outdegree of $v$

$u$ is a predecessor to $v$ if $(u,v) \in E$ — if it can be reached by travelling backwards.
Therefore $v$ has indegree $k$ if it has $k$ predecessors.

$$s_v = \max_{u \in \text{Predecessors}(v)} \left( s_u + \text{weight-edge}(u, v) \right)$$