Dynamic programming

Coin change problem

Input: a number P (price)

Output: a way of paying P using bills ($5, $1) or coins (1, 5, 10, 25 cents)

Objective: minimize the number of coins used (greedy approach)

$$8.37 = 5 + 1 + 1 + 1 + 0.25 + 0.10 + 0.01 + 0.01$$

8 bills/coins used

The Change Problem

Goal: Convert some amount of money ***M*** into given denominations, using the fewest possible number of coins

Input: An amount of money ***M***, and an array of ***d*** denominations $\mathbf{c} = (c_1, c_2, \ldots, c_d)$, in a decreasing order of value ($c_1 > c_2 > \ldots > c_d$)

Output: A list of *d* integers $i_1, i_2, \ldots, i_d$ such that

$$c_1 i_1 + c_2 i_2 + \ldots + c_d i_d = \boldsymbol{M}$$

and $i_1 + i_2 + \ldots + i_d$ is minimal

Change Problem: Example

Given the denominations 1, 3, and 5, what is the minimum number of coins needed to make change for a given value?

Value	1	2	3	4	5	6	7	8	9	10
Min # of coins	**1**		**1**		**1**					

Initialization: only one coin is needed to make change for the values 1, 3, and 5

Change Problem: Example

Given the denominations 1, 3, and 5, what is the minimum number of coins needed to make change for a given value?

Value	1	2	3	4	5	6	7	8	9	10
Min # of coins	**1**	**2**	**1**	**2**	**1**	**2**		**2**		**2**

However, two coins are needed to make change for the values 2, 4, 6, 8, and 10.

Change Problem: Example

Given the denominations 1, 3, and 5, what is the minimum number of coins needed to make change for a given value?

Value	1	2	3	4	5	6	7	8	9	10
Min # of coins	**1**	**2**	**1**	**2**	**1**	**2**	**3**	**2**	**3**	**2**

Lastly, three coins are needed to make change for the values 7 and 9

Change Problem: Recurrence

This example is expressed by the following recurrence relation:

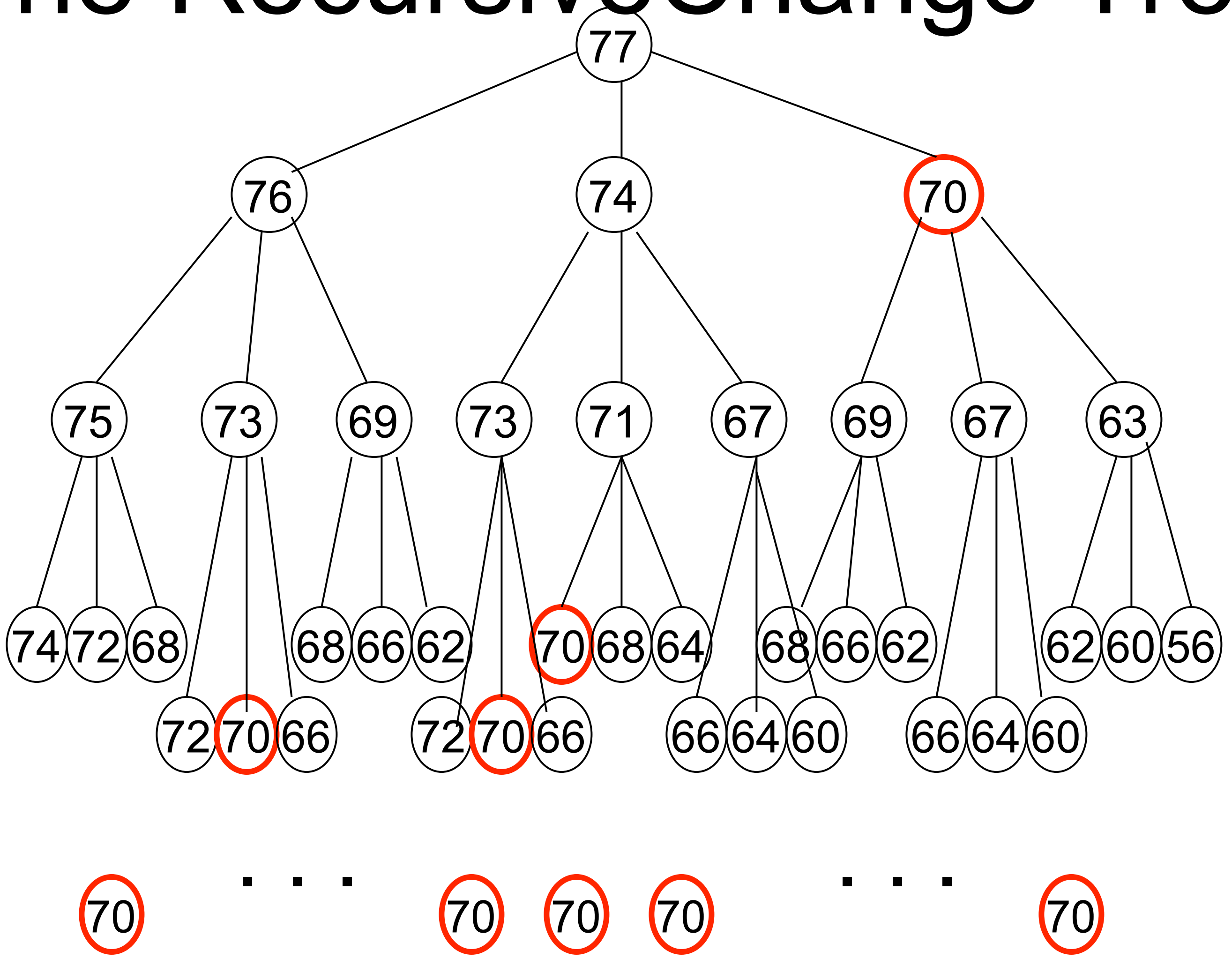
$$minNumCoins(M) = \textbf{min} \begin{cases} minNumCoins(M-1) + 1 \\ minNumCoins(M-3) + 1 \\ minNumCoins(M-5) + 1 \end{cases}$$

Change Problem: Recurrence (cont'd)

Given the denominations ***c***: c_1, c_2, …, c_d, the recurrence relation is:

$$minNumCoins(M) = \mathbf{min} \begin{cases} minNumCoins(M-c_1) + 1 \\ minNumCoins(M-c_2) + 1 \\ \dots \\ minNumCoins(M-c_d) + 1 \end{cases}$$

The RecursiveChange Tree



We can do better

- We’re re-computing values in our algorithm more than once
- Save results of each computation for 0 to ***M***
- This way, we can do a reference call to find an already computed value, instead of re-computing each time
- Running time ***M********d***, where ***M*** is the value of money and ***d*** is the number of denominations

The change problem: dynamic programming

DPChange($\boldsymbol{M}$,$\boldsymbol{c}$,$\boldsymbol{d}$)

$bestNumCoins_0 \leftarrow 0$

for $\boldsymbol{m} \leftarrow 1$ to $\boldsymbol{M}$

$bestNumCoins_m \leftarrow$ infinity

for $i \leftarrow 1$ to $\boldsymbol{d}$

if $\boldsymbol{m} \geq c_i$

if $bestNumCoins_{m - c_i} + 1 < bestNumCoins_m$

$bestNumCoins_m \leftarrow bestNumCoins_{m - c_i} + 1$

return $bestNumCoins_M$

DPChange: example

0
0

0	1
0	1

0	1	2
0	1	2

0	1	2	3
0	1	2	1

0	1	2	3	4
0	1	2	1	2

0	1	2	3	4	5
0	1	2	1	2	3

0	1	2	3	4	5	6
0	1	2	1	2	3	2

0	1	2	3	4	5	6	7
0	1	2	1	2	3	2	1

0	1	2	3	4	5	6	7	8
0	1	2	1	2	3	2	1	2

0	1	2	3	4	5	6	7	8	9
0	1	2	1	2	3	2	1	2	3

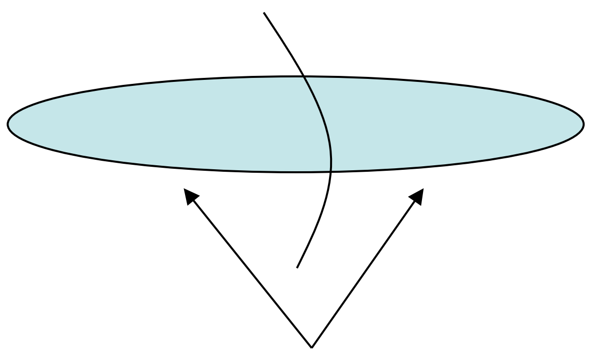
$\boldsymbol{c} = (1,3,7)$

$\boldsymbol{M} = 9$

DP: two key ingredients

- Two key ingredients for an optimization problem to be suitable for a dynamic-programming solution:

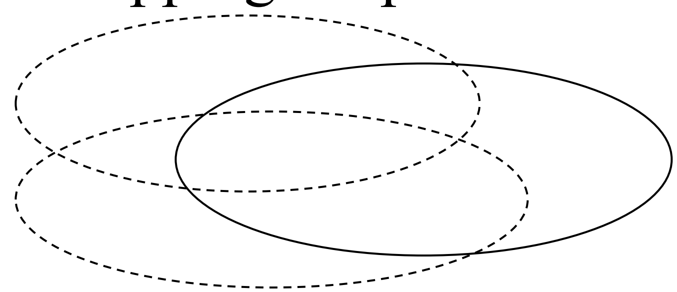
1. optimal substructures



Each substructure is optimal.

(Principle of optimality)

2. overlapping subproblems



Subproblems are dependent.

(otherwise, a divide-and-conquer approach is the choice.)

Three steps

- A DP algorithm has three basic steps:
  - The recurrence relation (for defining the value of an optimal solution);
  - The tabular computation (for computing the value of an optimal solution, while memoizing and avoiding recomputation);
  - The traceback (for delivering an optimal solution).

DPChange: example

0
0

0	1
0	1

0	1	2
0	1	2

0	1	2	3
0	1	2	1

0	1	2	3	4
0	1	2	1	2

0	1	2	3	4	5
0	1	2	1	2	3

0	1	2	3	4	5	6
0	1	2	1	2	3	2

0	1	2	3	4	5	6	7
0	1	2	1	2	3	2	1

0	1	2	3	4	5	6	7	8
0	1	2	1	2	3	2	1	2

0	1	2	3	4	5	6	7	8	9
0	1	2	1	2	3	2	1	2	3

$\boldsymbol{c} = (1,3,7)$

$\boldsymbol{M} = 9$

Aligning sequences with insertions and deletions

TGCATAT → ATCCGAT in 4 steps

TGCATAT → (insert A at front)

ATGCATAT → (delete 6th T)

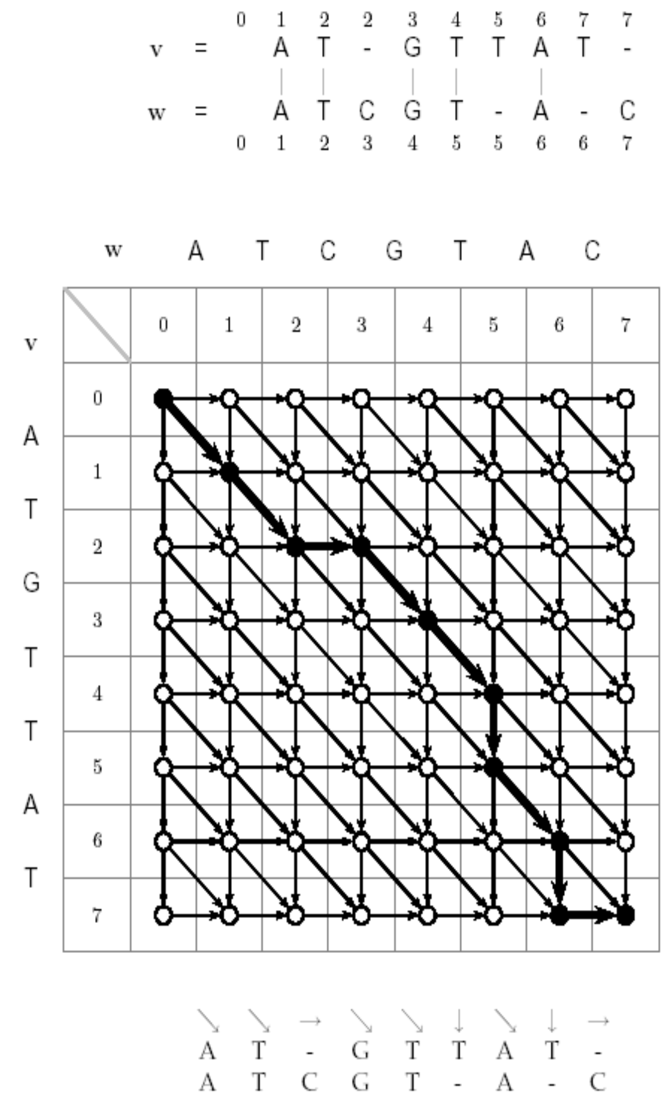
ATGCATA → (substitute G for 5th A)

ATGCGTA → (substitute C for 3rd G)

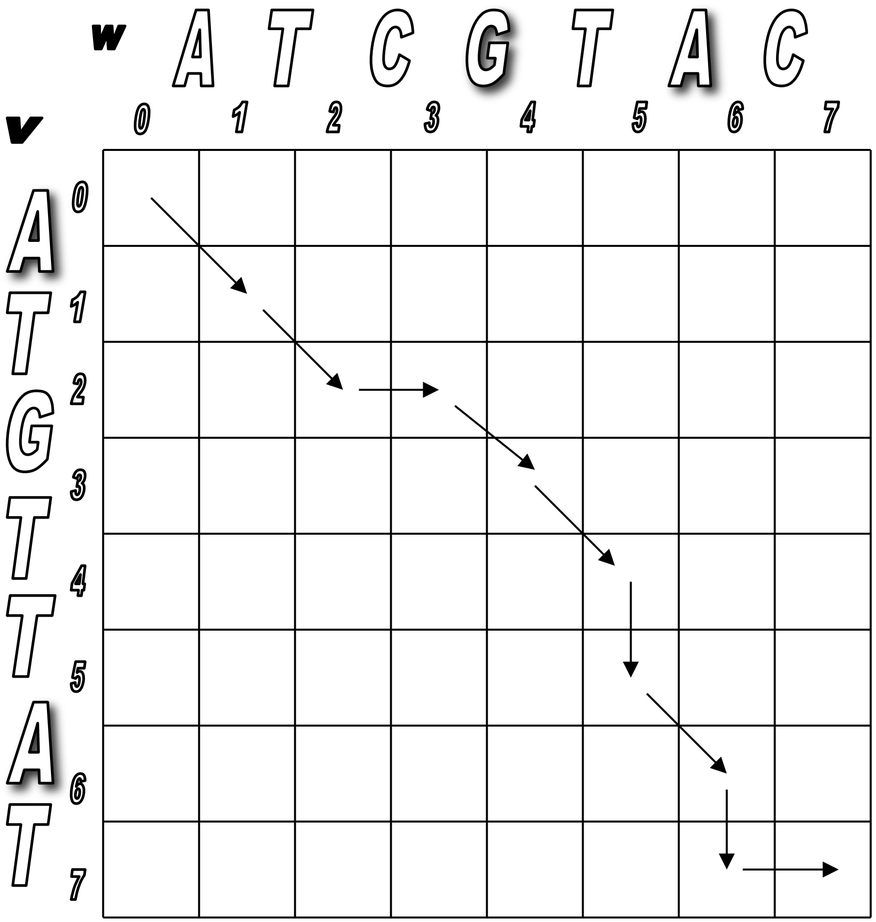
ATCCGAT (Done)

The alignment graph

– Every alignment path is from source to sink



Alignment as a path in the alignment graph

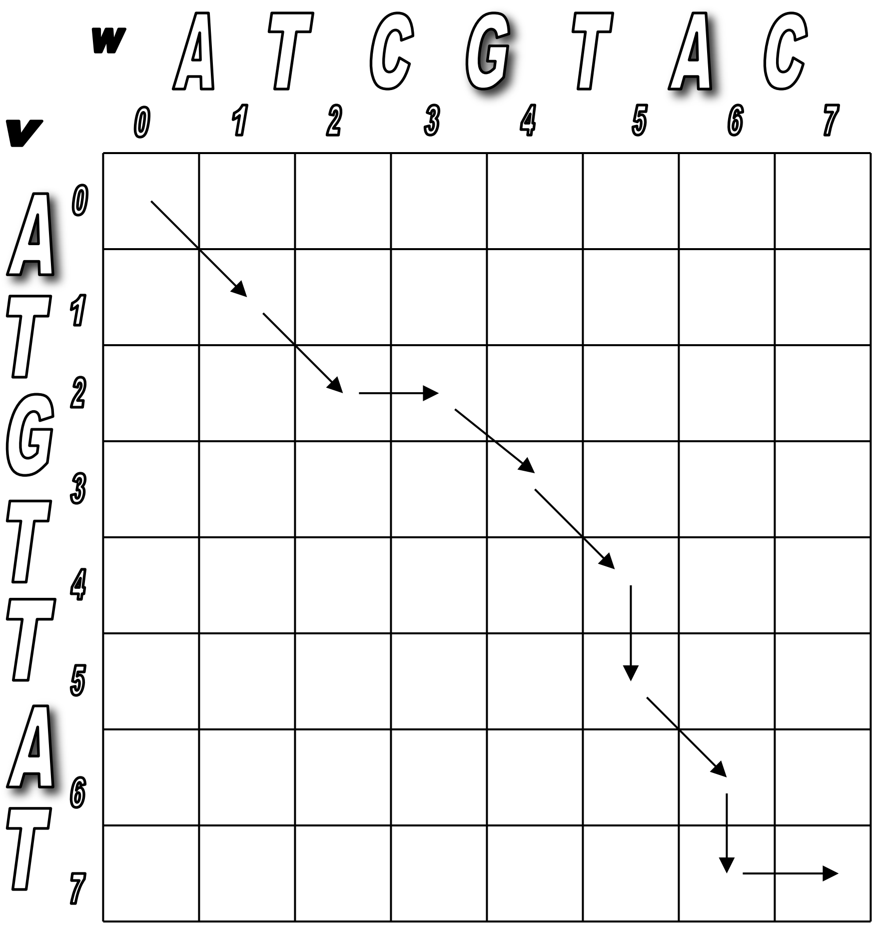


0	1	2	2	3	4	5	6	7	7
	A	T	_	G	T	T	A	T	_
	A	T	C	G	T	_	A	_	C
0	1	2	3	4	5	5	6	6	7

- Corresponding path -

(0,0) , (1,1) , (2,2), (2,3), (3,4), (4,5), (5,5), (6,6), (7,6), (7,7)

Alignment as a Path in the Edit Graph



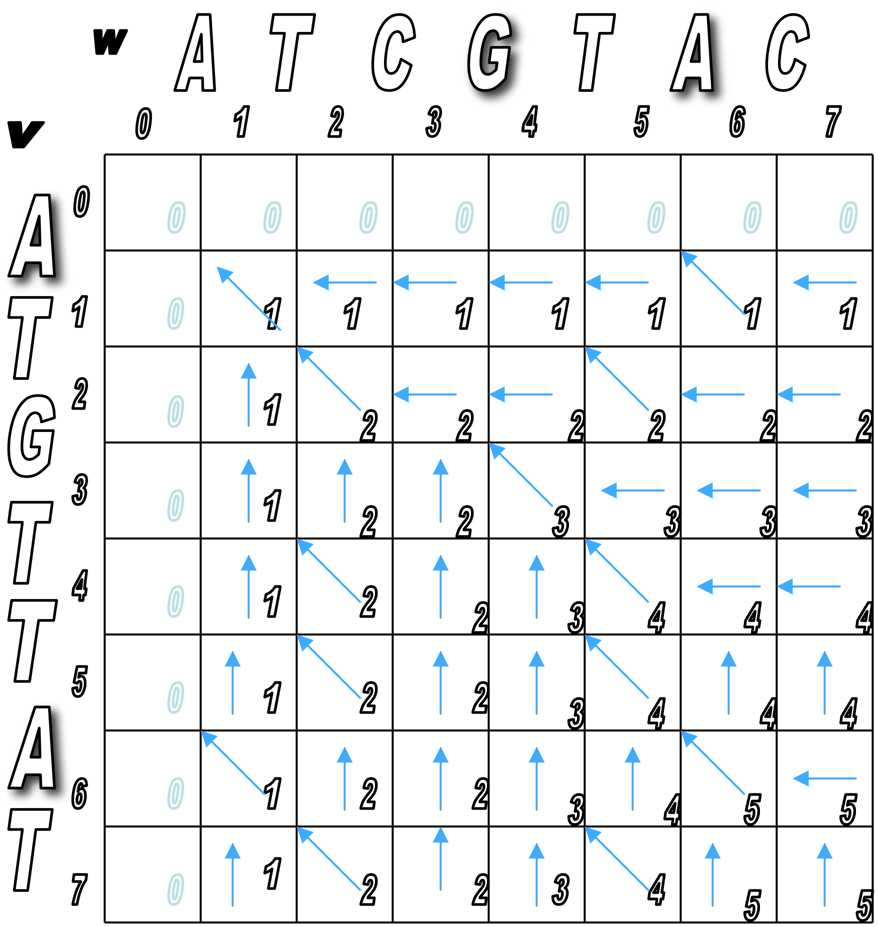
Every path in the edit graph corresponds to an alignment:

↘	↘	→	↘	↘	↓	↘	↓	→
A	T	-	G	T	T	A	T	-
A	T	C	G	T	-	A	-	C

Alignment: Dynamic Programming

$$s_{i,j} = \max \begin{cases} s_{i-1,\, j-1} + 1 \text{ if } v_i = w_j & \searrow \\ s_{i-1,\, j} & \downarrow \\ s_{i,\, j-1} & \rightarrow \end{cases}$$

Backtracking Example



Continuing with the dynamic programming algorithm gives this result.

Alignment: Backtracking

Arrows $\nwarrow$ $\leftarrow$ $\uparrow$ show where the score originated from.

$\uparrow$ if from the top

$\leftarrow$ if from the left

$\nwarrow$ if $v_i = w_j$

Dynamic programming: why it is working?

$$s_{i,j} = \min \begin{cases} s_{i-1,\, j-1} \text{ if } v_i = w_j \\ s_{i-1,\, j}+1 \\ s_{i,\, j-1}+1 \end{cases}$$

Longest common subsequence (LCS) – Alignment without mismatches

- Given two sequences

$$\boldsymbol{v} = v_1\, v_2 \ldots v_m \text{ and } \boldsymbol{w} = w_1\, w_2 \ldots w_n$$

- The LCS of ***v*** and ***w*** is a sequence of positions in

$$\boldsymbol{v}\text{:}\ 1 \leq i_1 < i_2 < \ldots < i_t \leq m$$

and a sequence of positions in

$$\boldsymbol{w}\text{:}\ 1 \leq j_1 < j_2 < \ldots < j_t \leq n$$

such that i_t*-th* letter *of* ***v*** *equals to* j_t*-letter of* ***w*** and ***t*** is maximal

Dynamic programming

$$s_{i,j} = \max \begin{cases} s_{i-1,\, j-1} + 1 \text{ if } v_i = w_j & \searrow \\ s_{i-1,\, j} & \downarrow \\ s_{i,\, j-1} & \rightarrow \end{cases}$$

From edit distance to sequence alignment

- The Edit Distance problem (or LCS) represents the simplest form of sequence alignment – allows only simple penalty for insertions, deletions and mismatches.
- In the LCS Problem, we scored 1 for matches and 0 for indels and mismatches
- For Edit distances, we scored 0 for matches and 1 for indels and mismatches
- Consider penalizing indels and mismatches with negative scores

Scoring Matrices

To generalize scoring, consider a (4+1) x(4+1) **scoring matrix** δ.

In the case of an amino acid sequence alignment, the scoring matrix would be a (20+1)x(20+1) size. The addition of 1 is to include the score for comparison of a gap character "-".

This will simplify the algorithm as follows:

$$s_{i,j} = max \begin{cases} s_{i-1,j-1} + \delta(v_i, w_j) \\ s_{i-1,j} + \delta(v_i, -) \\ s_{i,j-1} + \delta(-, w_j) \end{cases}$$

Scoring Indels: Naive Approach

- A fixed penalty σ is given to every indel:
  - $-\sigma$ for 1 indel,
  - -2σ for 2 consecutive indels
  - -3σ for 3 consecutive indels, etc.

Can be too severe penalty for a series of 100 consecutive indels

Affine Gap Penalties

- In nature, a series of *k* indels often come as a single event rather than a series of *k* single nucleotide events:

ATA__GC	*ATAG_GC*
ATATTGC	*AT_GTGC*

This is more likely.

Normal scoring would give the same score for both alignments

This is less likely.

Accounting for Gaps

- *Gaps*- contiguous sequence of spaces in one of the rows

- Score for a gap of length x is:

$$gap(x) = -(\rho + \sigma x)$$

where ρ >0 is the penalty for introducing a gap:

gap opening penalty

ρ will be large relative to σ*:*

gap extension penalty

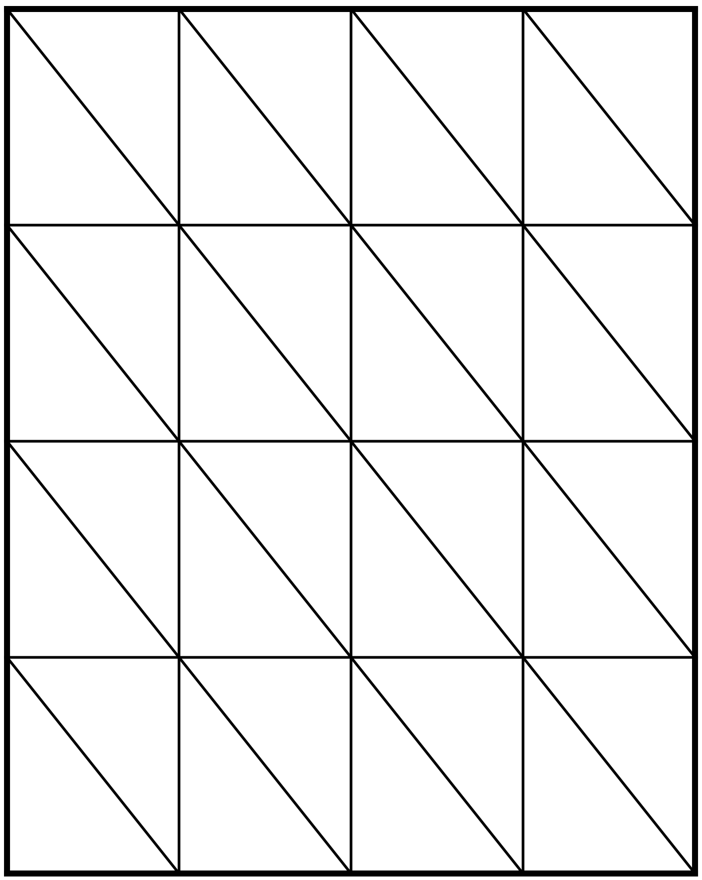
because you do not want to add too much of a penalty for extending the gap.

Can we still use the same recursion?

$$s_{i,j} = max \begin{cases} s_{i-1,j-1} + \delta(v_i, w_j) \\ s_{i-1,j} + \delta(v_i, -) \\ s_{i,j-1} + \delta(-, w_j) \end{cases}$$

$\delta(v_i, -) = \delta(-, w_j) = \rho + \sigma$?

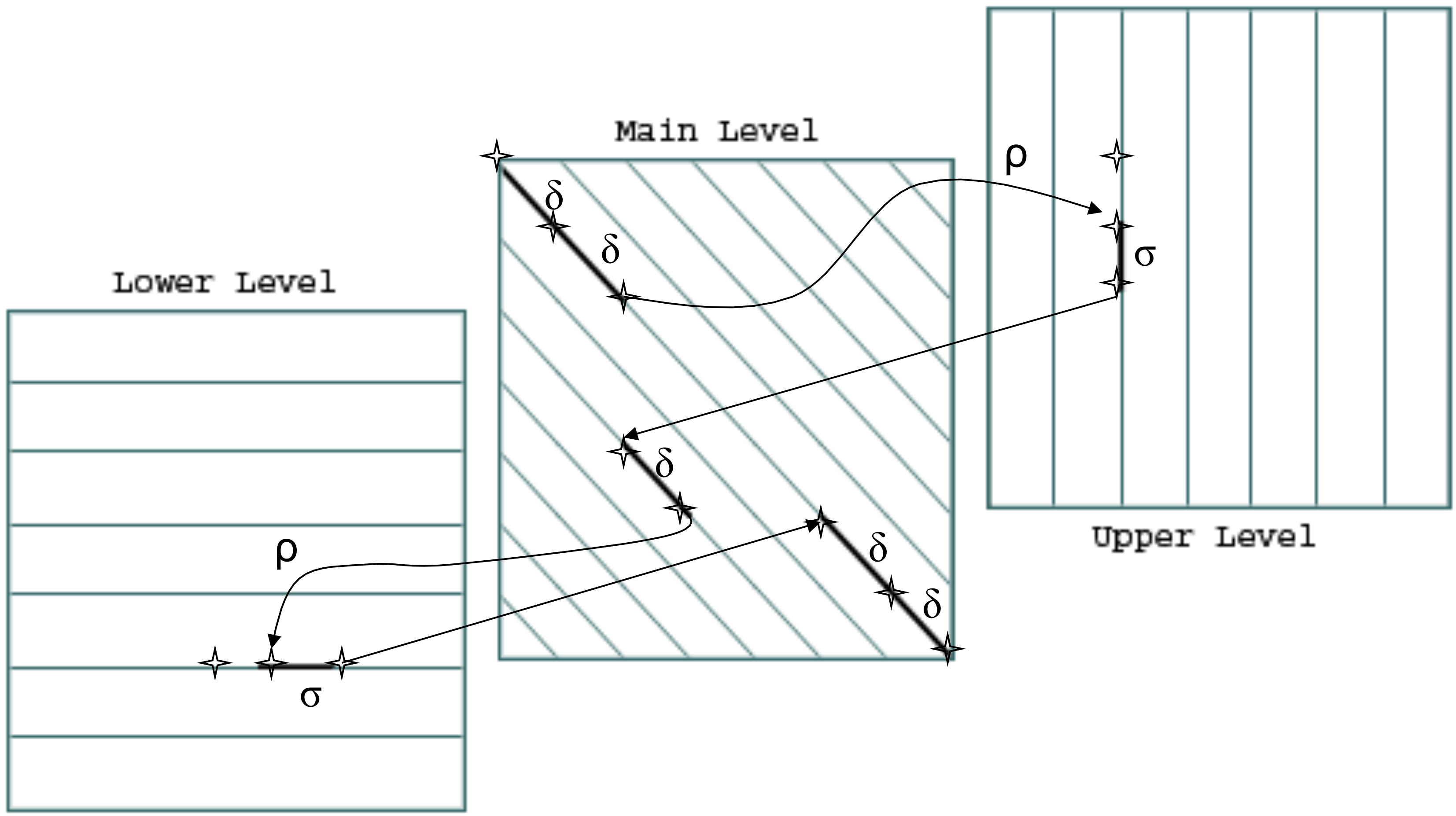
Affine gap penalties and alignment graph



To reflect affine gap penalties we have to add “long” horizontal and vertical edges to the edit graph. Each such edge of length *x* should have weight

$$-\rho - x * \sigma$$

Alignment graph in 3 Layers



Affine Gap Penalty Recurrences

$$\overset{\downarrow}{s}_{i,j} = \max \begin{cases} \overset{\downarrow}{s}_{i-1,j} - \sigma & \text{Continue Gap in } w \text{ (deletion)} \\ s_{i-1,j} - (\rho+\sigma) & \text{Start Gap in } w \text{ (deletion): from middle} \end{cases}$$

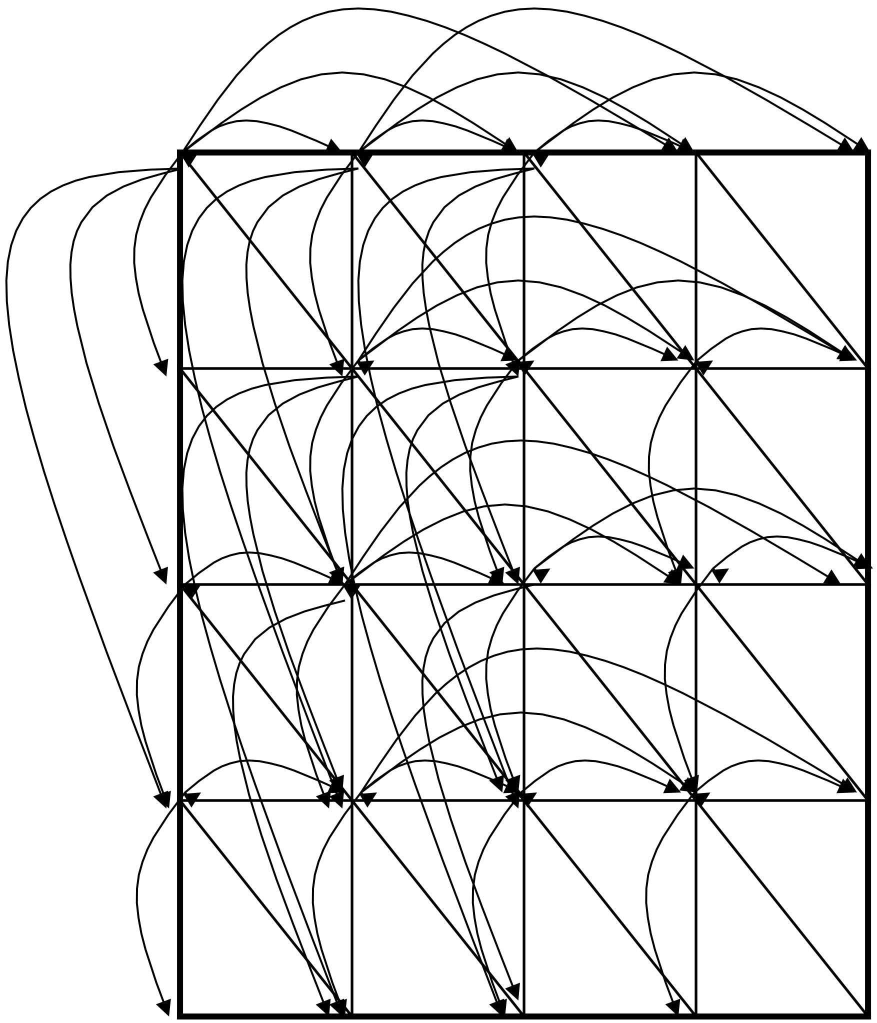
$$\vec{s}_{i,j} = \max \begin{cases} \vec{s}_{i,j-1} - \sigma & \text{Continue Gap in } v \text{ (insertion)} \\ s_{i,j-1} - (\rho+\sigma) & \text{Start Gap in } v \text{ (insertion):from middle} \end{cases}$$

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + \delta(v_i, w_j) & \text{Match or Mismatch} \\ \overset{\downarrow}{s}_{i,j} & \text{End deletion: from top} \\ \vec{s}_{i,j} & \text{End insertion: from bottom} \end{cases}$$

Arbitrary gap penalty

- Score for a gap of length *x* by gap(x)

“gap penalty” edges to the alignment graph



There are many such edges!

So the complexity increases from $O(n^2)$ to $O(n^3)$