Quiz

Mean: 25.5 / 30

Max: 30

SD: 3.9

Sequence similarity

DNA: From a computer scientist's viewpoint, DNA is a sequence of characters chosen from {A, C, G, T}.

- Human genome contains ~3 billion characters; an A4 paper contains 5 to 10 K characters, you need ~.5 million sheets of paper.

Given two DNA, biologists want to know how similar they are.

From the computational point of view, we first find the best way to align (pair-up) two sequences, then we can see how close they are.

Alignment and similarity

- An alignment : pairing up two strings character by character, possibly with space inserted.
- Example: ACCAATCC and AGCCATGC

```
ACCAATCC        A_CCAATCC
AGCCATGC        AGCCA_TGC
```

- 1st alignment: 5 positions matched; 3 mismatched

  2nd alignment: 6 positions matched; 3 mismatched
- What is the best alignment?

Similarity function

- A similarity function (score) δ specifies how much each match/mismatch/space contributes to the overall similarity.
- E.g., match: 2; mismatch: -1; character-space: -1.

	_	A	C	G	T
_		-1	-1	-1	-1
A	-1	2	-1	-1	-1
C	-1	-1	2	-1	-1
G	-1	-1	-1	2	-1
T	-1	-1	-1	-1	2

$\delta(C,G) = -1$

- Quality of an alignment = sum of similarity score over all positions.
  - Example:

ACCAATCC
AGCCATGC
score: 10 – 3 = 7

A_CCAATCC
AGCCA_TGC
score: 12 – 3 = 9.

Similarity function

- A more complicated similarity function.

	_	A	C	G	T
_		-.5	-.5	0	0
A	-.5	2	0.5	-1	-1
C	-.5	.5	4	-1	-1
G	0	-1	-1	3	-1
T	0	-1	-1	-1	2

The alignment problem

- With respect to a similarity function, an optimal alignment is an alignment with the maximum score.
- The alignment problem is to find the optimal alignment (also known as the global alignment problem) and its score.

Needleman-Wunsch algorithm

- Consider two strings S[1..n] and T[1..m].
- Definition:
  - V(S[1..i], T[1..j]) or V(i, j) = the score of the optimal alignment between the substrings S[1..i] and T[1..j]
- Basis:
  - $V(0, 0) = 0$
  - $V(0, j) = j\,\delta(\_, T[j])$
    - T[1..j] is matched with j spaces.
  - $V(i, 0) = i\,\delta(S[i], \_)$
    - S[1..i] is matched with i spaces.

Needleman-Wunsch algorithm

- Recurrence: For i > 0, j > 0
  - $$V(i,j) = \max \begin{cases} V(i-1,j-1) + \delta(S[i],T[j]) & \text{Match/mismatch} \\ V(i-1,j) + \delta(S[i],\_) & \text{Add space to T} \\ V(i,j-1) + \delta(\_,T[j]) & \text{Add space to S} \end{cases}$$

```
S   xxx…xx     xxx…xx    xxx…x_
         |          |         |
T   yyy…yy     yyy…y_    yyy…yy
```

Example (I)

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

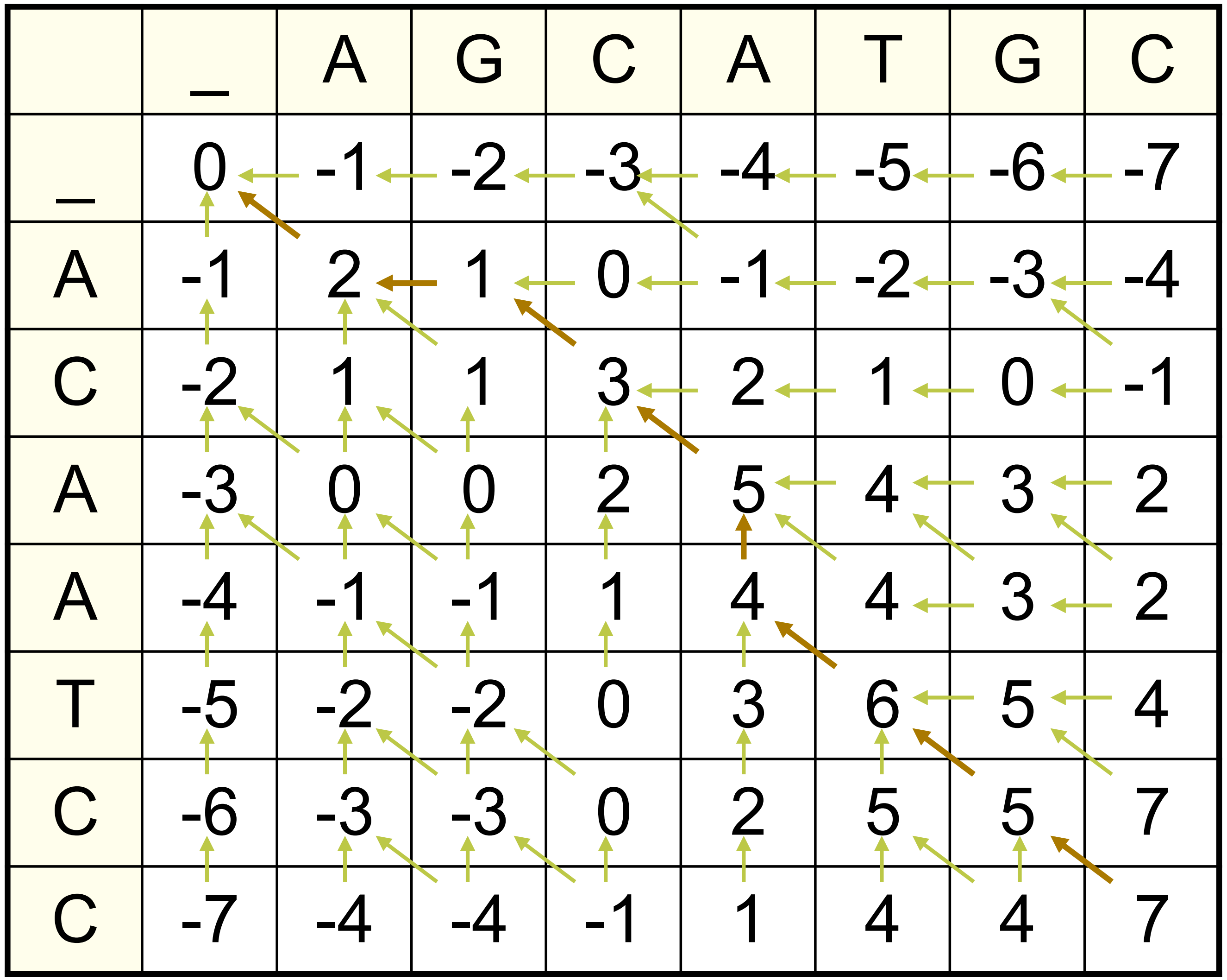
PS. match: 2; mismatch: -1; character-space: -1.

Example (II)

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

PS. match: 2; mismatch: -1; character-space: -1.

Backward tracing



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

Analysis

- We need to fill in all entries in the table, which is of size $n \times m$.
- Each entry can be computed in O(1) time.
- Time complexity = O(nm)
- Space complexity = O(nm)

Problem on Space (memory)

- Note that the dynamic programming requires a lot of space O(mn).
- When we compare two very long sequences, space may be the limiting factor.
  - E.g., each sequence of length 100K.   10G words
- Can we solve the string alignment problem in linear space ?
  - O(n+m) words.

Suppose we don't recover the alignment

- The DP table can be filled in row by row.
- If we are only interested in computing the optimal alignment score,
  - store the two current rows of the table, and
  - the space complexity becomes O(m).

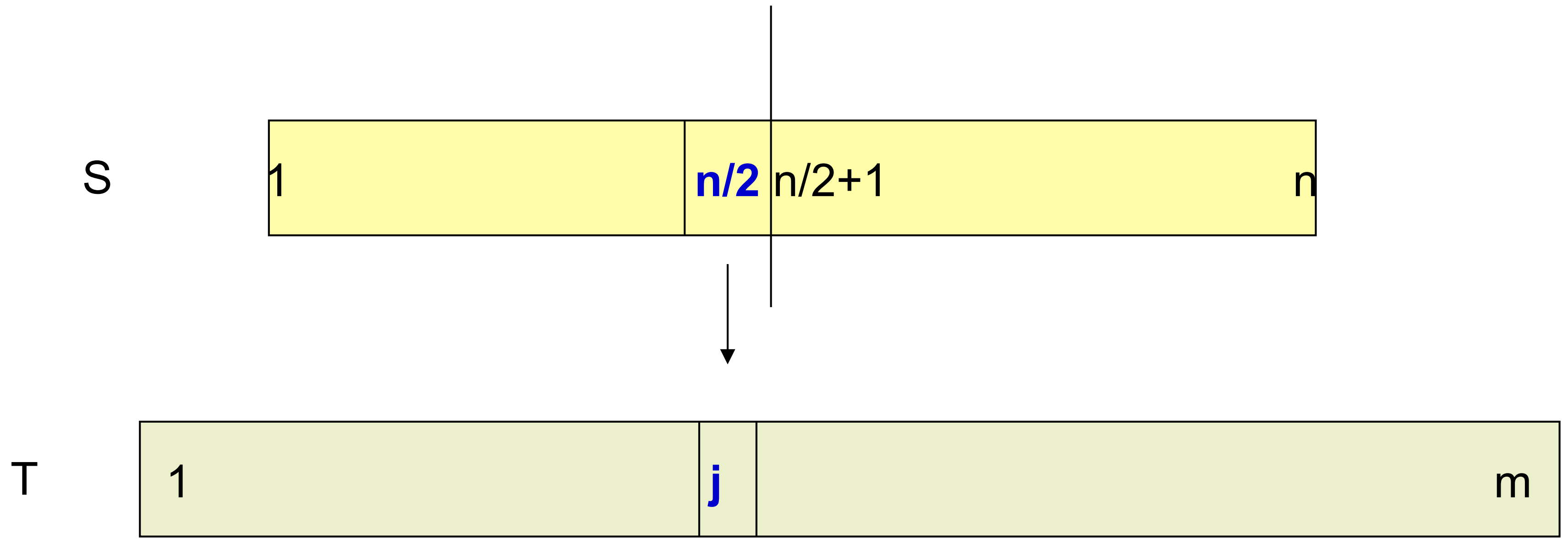
- Summary. Given strings S[1..n] and T[1..m], we can compute in O(m) space the followings.
  - V[n,m]
  - V[n,1], V[n,2], …, V[n,m]

V[n,1]	V[n,2]	V[n,3]	…	V[n,m]

Recover the mid-point using O(n+m) space

Let us look at a simple problem first.

In an optimal alignment, which character of T is S[n/2] aligned to ?



How to find the mid-point alignment

Fact. $V(S[1..n], T[1..m]) =$

$$\max_{0 \le j \le m} \left\{ V(S[1..\tfrac{n}{2}], T[1..j]) + V(S[\tfrac{n}{2}+1..n], T[j+1..m]) \right\}$$

1. Re-compute V(S[1..n/2], T[1..j]) for all j.
2. Re-compute V(S[n/2 + 1 .. n], T[j+1 .. m]) for all j.
3. Determine which j maximizes the above sum.

Step 1: DP again

Fact.
$$\mathsf{V}(\mathsf{S}[1..n],\mathsf{T}[1..m]) = \max_{0\le j\le m}\left\{\mathsf{V}(\mathsf{S}[1..\tfrac{n}{2}],\mathsf{T}[1..j]) + \mathsf{V}(\mathsf{S}[\tfrac{n}{2}+1..n],\mathsf{T}[j+1..m])\right\}$$

1. Cost-only dynamic programming for the first half.
   - Input: S[1..n/2], T[1..m]
   - Output: V(S[1..n/2], T[1..j]) for **all** 0 ≤ j ≤ m

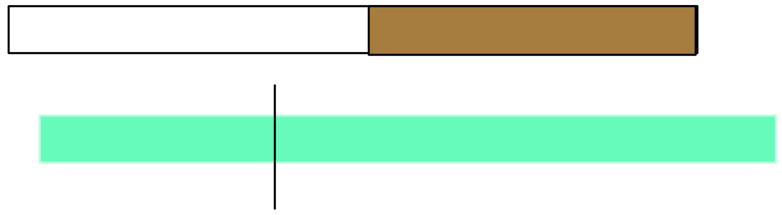
Step 2 is non-trivial

Fact.
$$V(S[1..n],T[1..m]) = \max_{0\le j\le m}\left\{V(S[1..\tfrac{n}{2}],T[1..j]) + V(S[\tfrac{n}{2}+1..n],T[j+1..m])\right\}$$

1. Cost-only dynamic programming for the first half.
   - Input: S[1..n/2], T[1..m]
   - Output: V(S[1..n/2], T[1..j]) for all $0 \le j \le m$.
2. How to find V(S[n/2+1..n], T[j+1..m]) for all j ?
   - Input: S[n/2+1..n], T[1..m]
   - DP: V(S[n/2+1..n], T[1..j]) for all $0 \le j \le m$.

Step 2: Reverse the string

1. V(S[1..n/2], T[1..j]) for all 1 ≤ j ≤ m.
2. Cost-only dynamic programming for the reverse of the second half.
   - Input:
     - S'[1..n/2] = reverse of S[n/2+1..n],
     - T'[1..m] = reverse of T[1..m]
     - Fact: **V**(S[n/2+1..n], T[j..m]) = **V**(S'[1..n/2], T'[1..j])
   - Compute **V**(S'[1..n/2], T'[1..j]) for **all** 0 ≤ j ≤ m.
   - Output: **V**(S'[1..n/2], T'[1..j]) for **all** 0 ≤ j ≤ m.
3. Determine which j maximizes the above sum.



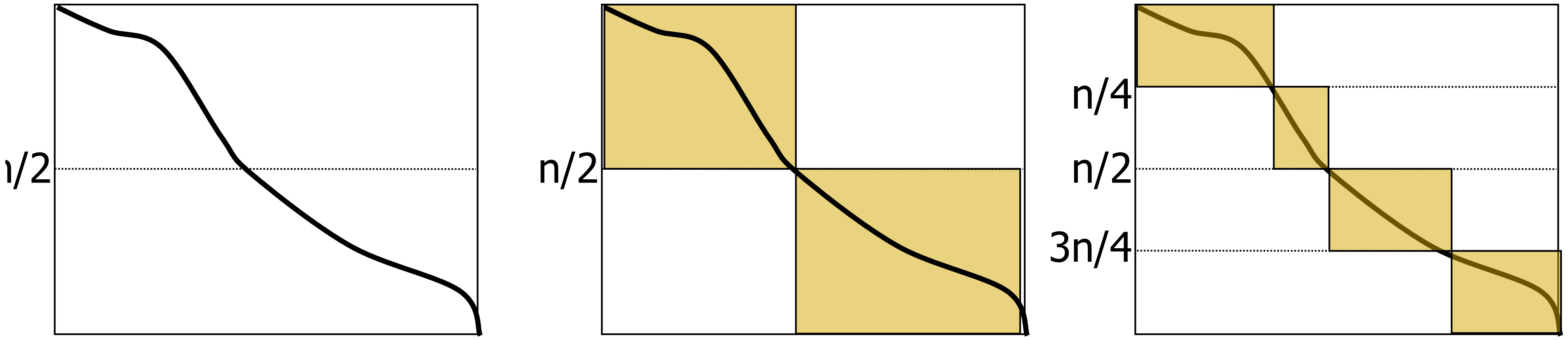
Time

- Time for finding mid-point:
  - Step 1 takes O(n/2 m) time
  - Step 2 takes O(n/2 m) time
  - Step 3 takes O(m) time.
  - In total, O(nm) time.

Recover the alignment in O(n+m) space?

Yes. By recursion. Idea:

1. Based on the cost-only algorithm, find the mid-point of the alignment.
2. Divide the problem into two halves.
3. Recursively deduce the alignments for the two halves.



Time Analysis

- Time for finding mid-point:
  - O(nm) time (cnm for some constant c)
- Let T(n, m) be the time for recovering the alignment.
- T(n, m) = time for finding mid-point alignment + time for solving the two subproblems

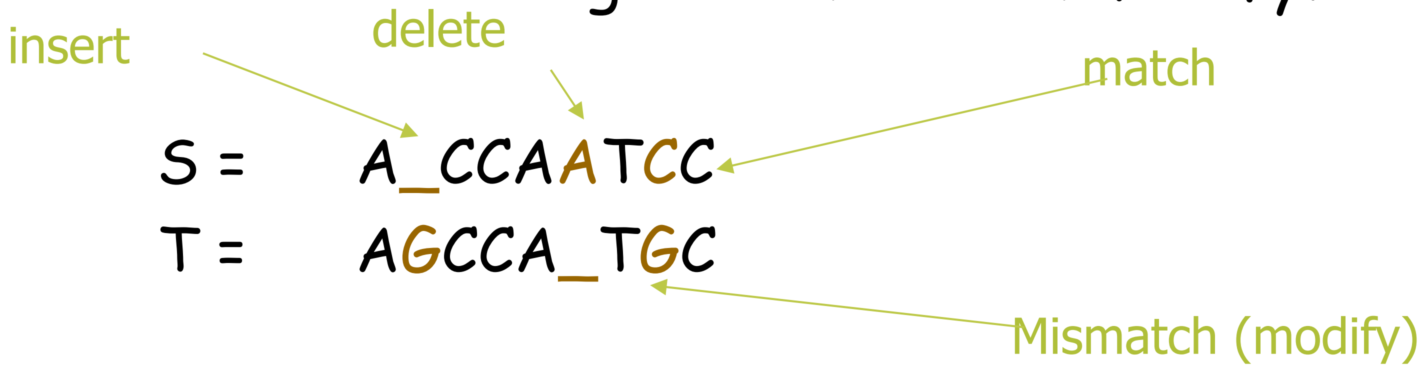
  $\leq$ c nm + T(n/2, j) + T(n/2, m-j) for some j.
  - T(n/2, j) $\leq$ c (n/2) j + T(n/4,j') + T(n/4,j-j')
  - T(n/2, m-j) $\leq$ c (n/2) (m – j) + T(n/4, j'') + T(n/4, m-j+j'')
  - T(n,m) $\leq$ cnm + cmn/2 + cmn/4 + …
- Thus, time complexity = T(n, m) $\leq$ 2c nm.

Space analysis

- Working memory for finding mid-point takes O(m) space.
- Once we find the mid-point, we can free the working memory.
- Thus, in each recursive call, we only need to store the alignment path.
- Observe that the alignment subpaths are disjoint, the total space required is O(n+m).

Edit distance

- Classic DP problem: Edit distance (S,T) = the fewest number of insert/delete/modify operations to transform S to T.
- The alignment of two strings S and T also defines a way to edit S into T using insert/delete/modify.



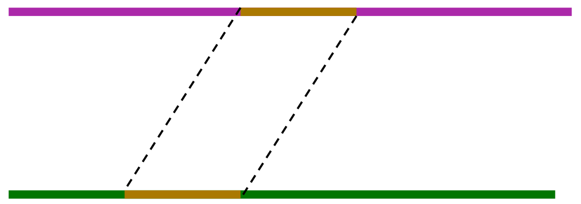
```
S =     A_CCAATCC
T =     AGCCA_TGC
```

The above alignment involves

- 1 modify (C to G),
- 1 insert (adding a space in S to match "G" of T),
- 1 delete ("A" matches a space of T).

- Edit distance is a special case of the alignment problem. (why ?)

Local Alignment



- Input: two strings (DNA) S and T.
- To compute: the most similar substrings of S & T (i.e., substrings A and B whose alignment score is maximized over all possible pairs of substrings)
- Intuitively, most similar substrings -> a common gene

Brute-force solution

- Algorithm:

```
For every substring A of S,
    For every substring B of T,
        Compute the global alignment of A and B
Return the pair (A, B) with the highest score
```

- Time complexity:
  - There are n^2 choices of A and m^2 choices of B.
  - The global alignment of A and B can be computed in $O(nm)$ time.
  - In total, $O(n^3m^3)$ time
- Can we do better ?

Some background

- X is a suffix of S[1..n] if X=S[k..n] for some $k \geq 1$
- X is a prefix of S[1..n] if X=S[1..k] for some $k \leq n$
- E.g.,
  - Consider S[1..7] = ACCGATT
  - ACC is a prefix of S, GATT is a suffix of S
  - Empty string is a prefix and suffix of S

Dynamic programming for local alignment

- Define V(i, j) to be the maximum score of the (global) alignment of A and B over
  - all suffixes A of S[1..i] and
  - all suffixes B of T[1..j]

- Then, the score of local alignment is
  - $\max_{i,j} V(i, j)$

Example

- S = ACCAATCC and T = AGCCATGC
- S[1,4] = ACCA; T[1,6] = AGCCAT

  S[1,4] has 4 suffixes, and T[1,6] has 6 suffixes.

- V(4, 6) is the maximum score of the global alignment of the following 24 pairs.

  S[x,4]: T[y,6], where x = 1 to 4, and y = 1 to 6.

Smith-Waterman algorithm

- Basis:
  - V(i, 0) = V(0, j) = 0
- Recursion for i>0 and j>0:
  -

$$V(i,j) = \max \begin{cases} 0 & \text{Align empty strings} \\ V(i-1, j-1) + \delta(S[i], T[j]) & \text{Match/mismatch} \\ V(i-1, j) + \delta(S[i], \_) & \text{Add space to T} \\ V(i, j-1) + \delta(\_, T[j]) & \text{Add space to S} \end{cases}$$

Example (I)

- Score for match = 2
- Score for matching with space, mismatch = -1

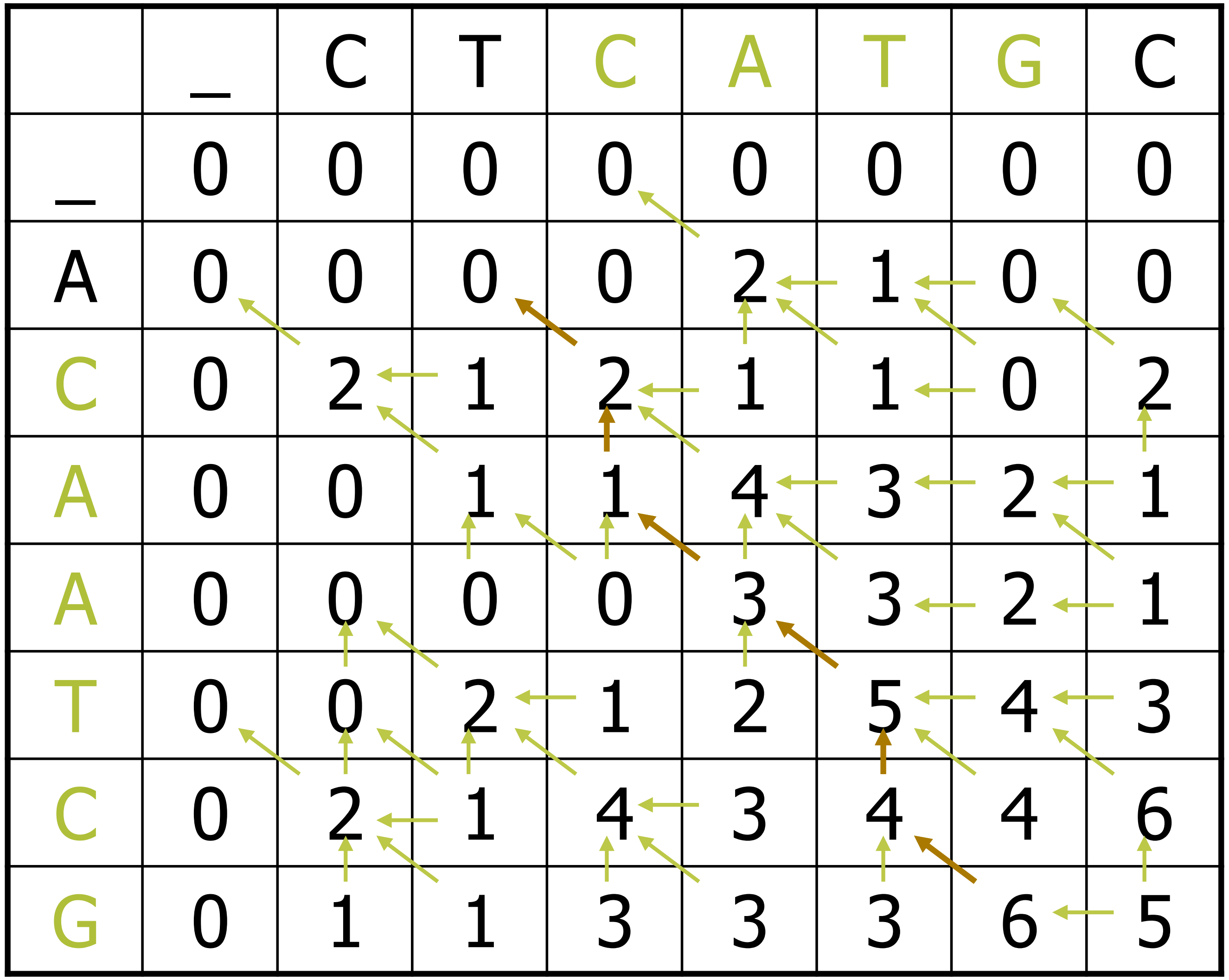
	_	C	T	C	A	T	G	C
_	0	0	0	0	0	0	0	0
A	0							
C	0							
A	0							
A	0							
T	0							
C	0							
G	0							

Example (II)

	_	C	T	C	A	T	G	C
_	0	0	0	0	0	0	0	0
A	0	0	0	0	2	1	0	0
C	0	2	1	2	1	1	0	2
A	0	0	1	1	4	3	2	1
A	0	0	0	0	3	3	2	1
T	0	0	2	1	2			
C								
G								

Example (III)

```
C_AT_G
CAATCG
```



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

Analysis

- We need to fill in all entries in the table.
- Each entry can be computed in O(1) time.
- Finally, finding the entry with the maximum value.
- Time complexity = O(nm)
- Space complexity = O(nm)