**Alignment with Affine Gap Penalties Problem**

*Find a highest-scoring global alignment between two strings (with affine gap penalties).*

**Input:** A match score *m*, a mismatch penalty *μ*, a gap opening penalty *σ*, a gap extension penalty *ε*, and two DNA strings *s* and *t*.

**Output:** The maximum alignment score of an alignment between *s* and *t* (using affine gap penalties) followed by an alignment achieving this maximum score.

A *gap* is a contiguous sequence of spaces in a row of an alignment. One way to score gaps more appropriately is to define an *affine penalty* for a gap of length *k* as *σ + ε* · (*k* − 1), where *σ* is the *gap opening penalty*, assessed to the first symbol in the gap, and *ε* is the *gap extension penalty*, assessed to each additional symbol in the gap. We typically select *ε* to be smaller than *σ* so that the affine penalty for a gap of length *k* is smaller than the penalty for *k* independent single-nucleotide indels (*σ · k*).

**Input Format.** The first line of the input contains *m* followed by *μ* followed by *σ* followed by *ε* (separated by spaces), the second line of the input contains a DNA string *s*, and the third line of the input contains a DNA string *t*.

**Output Format.** The first line of the output should contain the maximum score of an alignment between *s* and *t* using affine gap penalties, and the next two lines should contain an alignment achieving this maximum score. Specifically, the second line should contain *s* with gaps placed appropriately, and the third line should contain *t* with gaps placed appropriately.

**Constraints.** |*s*| ≤ 1,000; |*t*| ≤ 1,000

**SAMPLE DATASET:**

Input:

1 3 2 1

GA

GTTA

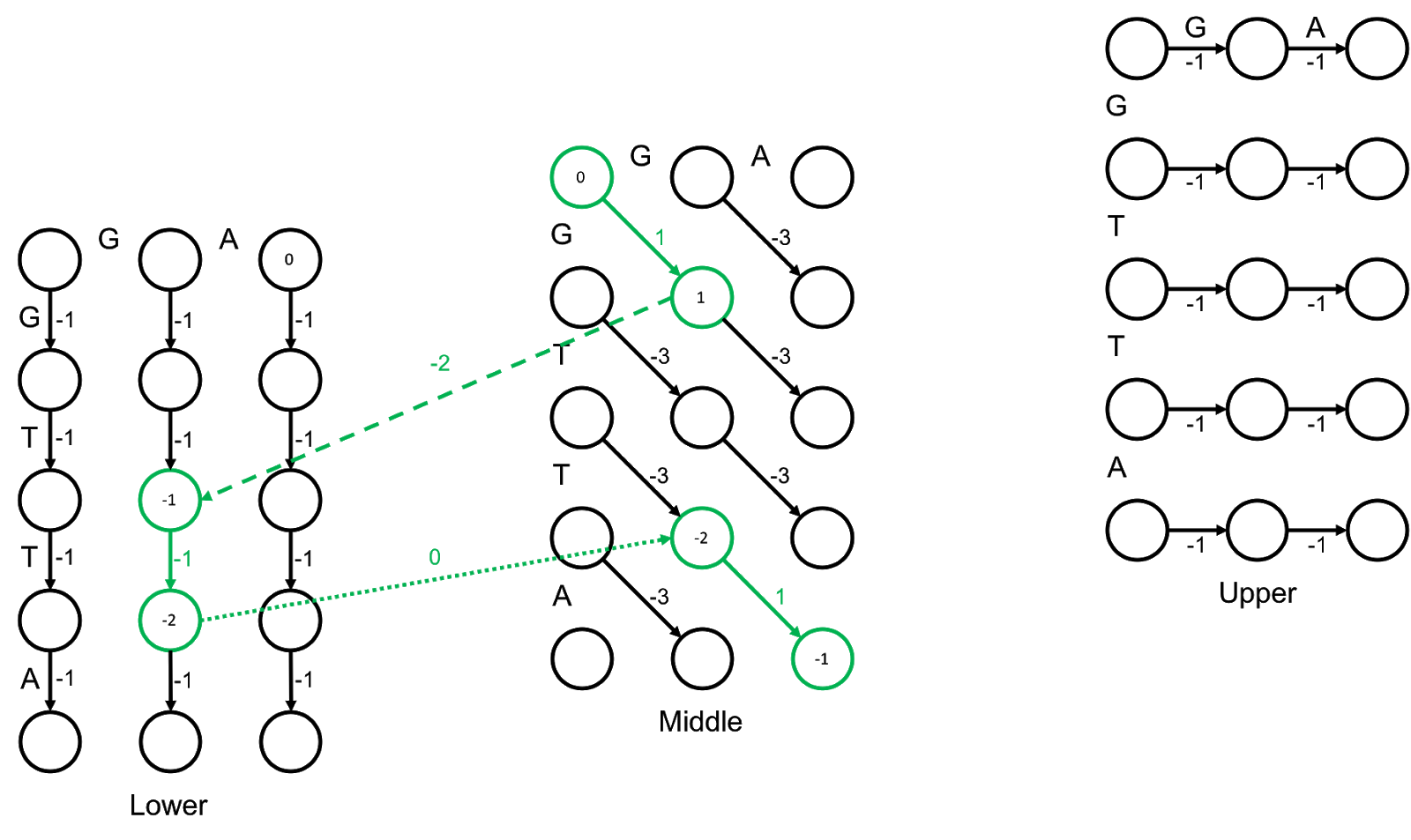
Output:

-1

G--A

GTTA

The highest score of an alignment (using affine gap penalties) between GA and GTTA is -1, and the above alignment achieves this maximum score.



**TEST DATASET 1:**

Input:

1 5 3 1

TTT

TT

Output:

-1

TTT OR TTT OR TTT

-TT T-T -TT

This test makes sure that your code is correctly parsing the gap opening and gap extension penalties. If your code swaps the values for the extension and opening penalties then your score will likely be 1 instead of the correct value of -1. The reconstructed alignment should not change even if you mix up the gap opening and gap extension penalties.

**TEST DATASET 2:**

Input:

1 5 5 1

GAT

AT

Output:

-3

GAT

-AT

This test makes sure that your code is implementing global alignment with affine gap penalties instead of fitting, overlap, or local alignment with affine gap penalties. All other types of alignment will simply align the two AT substrings and report a score of 2. Be sure that your implementation is of **global** alignment.

**TEST DATASET 3:**

Input:

1 5 2 1

CCAT

GAT

Output:

-3

-CCAT OR CC-AT

G--AT --GAT

This test makes sure that your *upper* and *lower* matrices are correctly initialized. Be especially careful about your initialization of the first row of the *lower* matrix and the first column of the *upper* matrix. Depending on your backtracking implementation it is possible that you will get the correct alignment reported despite having an incorrect score. This is likely due to an issue in your initialization of the *upper* and *lower* matrices. If you consistently get a score of -2 instead of the correct -3 it is possible that your code is incorrectly considering gaps that span across the two strings as one gap. This is not the case; there should be two gap opening penalties in the alignment for this dataset.

**TEST DATASET 4:**

Input:

1 2 3 2

CAGGT

TAC

Output:

-8

CAGGT

TAC--

This test makes sure that your code can handle a gap extension penalty that isn’t equal to one. If your output doesn’t match the correct output it’s likely that your implementation relies on the gap extension penalty being equal to one. Since all previous datasets set the gap extension penalty to one your code could have passed all previous tests without properly using the input to set the gap extension penalty.

**TEST DATASET 5:**

Input:

2 3 3 2

GTTCCAGGTA

CAGTAGTCGT

Output:

-8

--GT--TCCAGGTA

CAGTAGTC---GT-

This test makes sure that your code can handle inputs in which the two strings are the same length. If your output doesn’t match the correct output make sure that your code doesn’t make any assumptions about the lengths of the input strings. Since no previous dataset contained two strings with the same length your implementation could have passed all previous tests without handling the case where the two strings are the same length.

**TEST DATASET 6:**

Input:

1 3 1 1

AGCTAGCCTAG

GT

Output:

-7

AGCTAGCCTAG

-G-T-------

This test makes sure that your code can handle inputs in which the strings vary drastically in length. If your output doesn’t match the correct output make sure that your implementation doesn’t make any assumptions about the lengths of the strings. Make sure that your three dynamic programming matrices are assigned the correct dimensions given the input strings.

**TEST DATASET 7:**

Input:

2 1 2 1

AA

CAGTGTCAGTA

Output:

-7

-------A--A

CAGTGTCAGTA

This dataset checks that your code can handle inputs in which the two strings to be aligned are different lengths. This dataset is similar to Test Dataset 6 except that in this dataset string *s* is shorter than string *t*.

**TEST DATASET 8:**

Input:

5 2 15 5

ACGTA

ACT

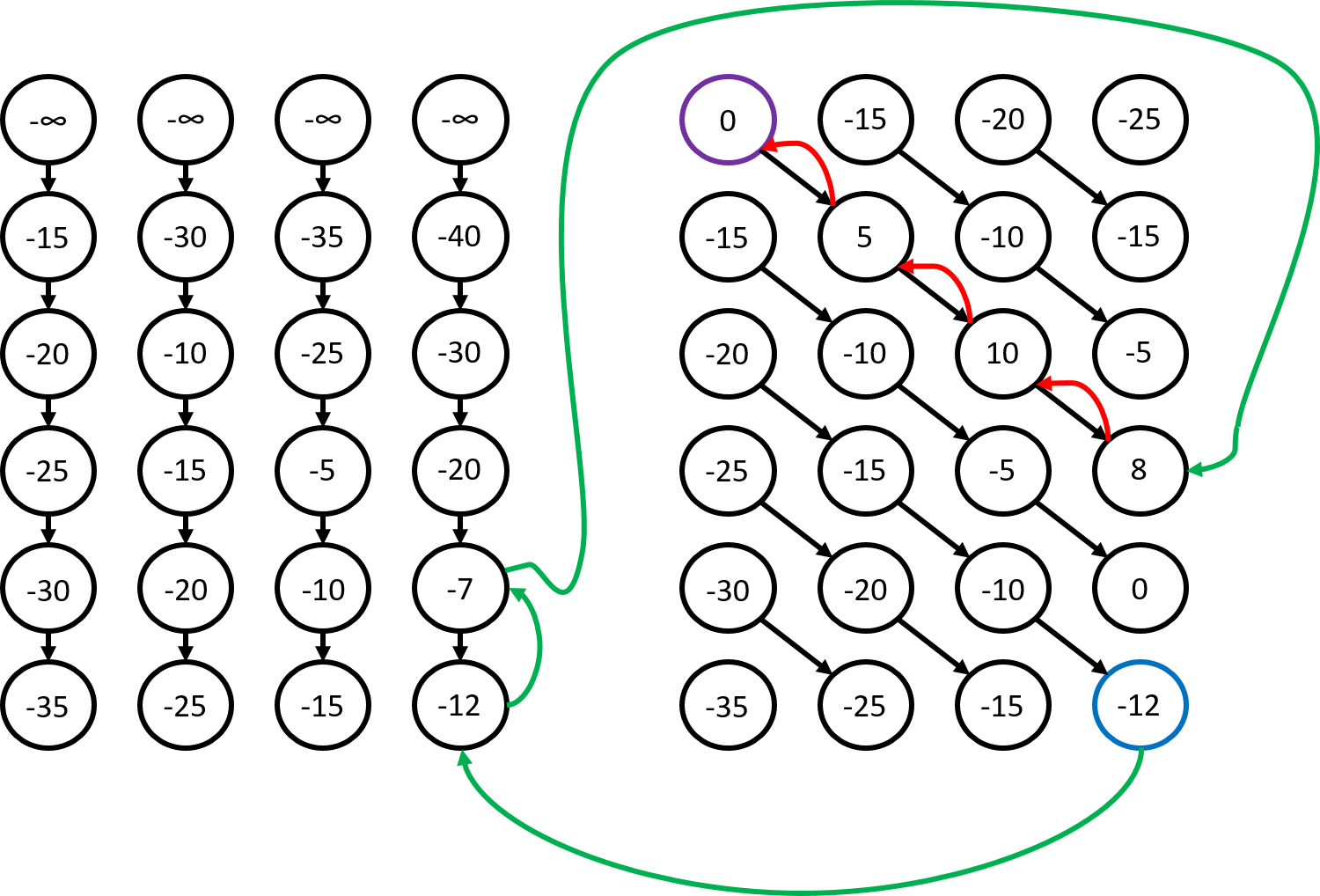
Output:

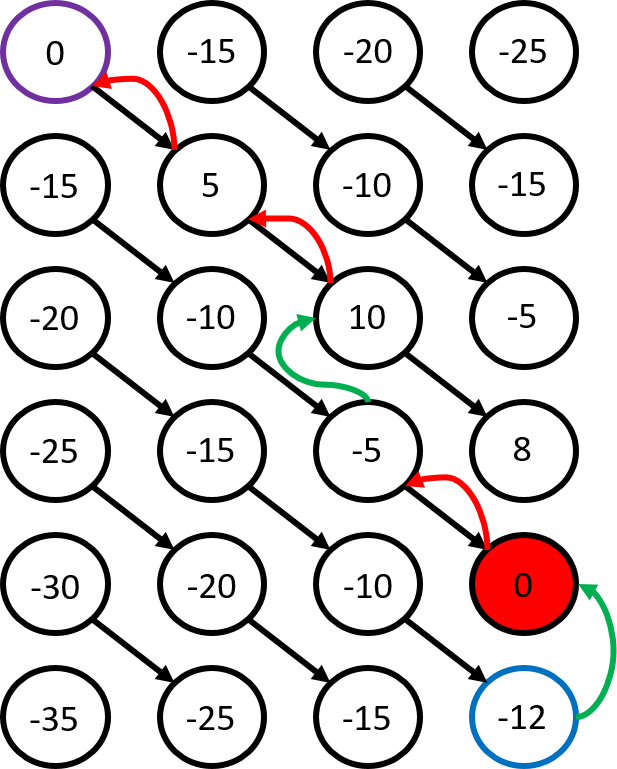
-12

ACGTA

ACT--

This dataset checks that your code is actually using three distinct matrices to reconstruct the alignment. It may be tempting to reconstruct the alignment using only the *middle* matrix but that could lead to subtle errors. If the last A character in string *s* was not present the ideal alignment would match the T characters. Once the last A character is added mismatching the T in string *t* with the G in string *s* yields a higher final score. If your implementation only uses one matrix then you are likely assuming that knowing if a gap is being initialized or extended is sufficient for scoring. Consider the figure below. In the first representation we have the *lower* matrix on left and the *middle* matrix on the right. Starting at the node outlined in blue and ending at the node outlined in purple we can backtrack through both matrices and end up with the correct reconstruction. However, if we only store a backtracking value indicating direction we immediately run into an issue. In the second figure only the *middle* matrix is shown. We know that we need to move upward from the blue-outlined node, but at the node shaded red our backtracking value would suggest a diagonal edge. In other words, using only one matrix to backtrack is not sufficient for every case.



****