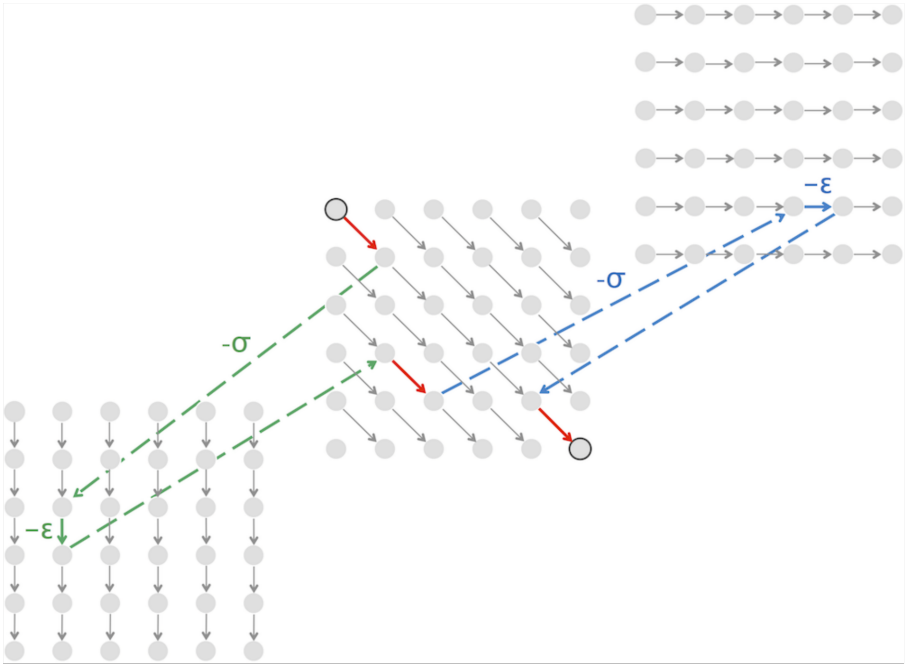
5J Align Two Strings Using Affine Gap Penalties

Alignment with Affine Gap Penalties Problem
Find the highest-scoring alignment between two strings (with affine gap penalties).

Input: Two DNA strings, a match reward, a mismatch penalty, a gap opening penalty, and a gap extension penalty.
Output: A highest-scoring global alignment with affine gap penalties of these two strings (with respect to the scoring parameters) and its score.



Formatting

Input: Four space-separated integers representing the match reward, mismatch penalty, gap opening penalty, and gap extension penalty respectively, followed by two space-separated DNA strings v and w.
Output: The maximum global alignment score of v and w using the scoring function with affine gap penalties and an alignment of v and w achieving this maximum score (if multiple global alignments with affine gap penalties achieving the maximum score exist, you may return any one).

Constraints

- The lengths of v and w will be between 1 and 10^2
- Both v and w will be DNA strings.

Test Cases

Case 1

Description: The sample dataset is not actually run on your code.

Input:

```
1 3 2 1
GA
GTTA
```

Output:

```
-1
G--A
GTTA
```

Case 2

Description: 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.

Input:

```
1 5 3 1
TTT
TT
```

Output:

```
-1
TTT
TT-
```

Case 3

Description: 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.

Input:

```
1 5 5 1
GAT
AT
```

Output:

```
-3
GAT
-AT
```

Case 4

Description: 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.

Input:

```
1 5 2 1
CCAT
GAT
```

Output:

```
-3
-CCAT
G--AT
```

Case 5

Description: 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.

Input:

```
1 2 3 2
CAGGT
TAC
```

Output:

```
-8
CAGGT
TAC--
```

Case 6

Description: 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.

Input:

```
2 3 3 2
GTTCCAGGTA
CAGTAGTCGT
```

Output:

```
-8
--GT--TCCAGGTA
CAGTAGTC---GT-
```

Case 7

Description: 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.

Input:

```
1 3 1 1
AGCTAGCCTAG
GT
```

Output:

```
-7
AGCTAGCCTAG
-G-T-------
```

Case 8

Description: 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 7 except that in this dataset string v is shorter than string w.

Input:

```
2 1 2 1
AA
CAGTGTCAGTA
```

Output:

```
-7
-------A--A
CAGTGTCAGTA
```

Case 9

Description: 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 v was not present the ideal alignment would match the "T" characters. Once the last "A" character is added mismatching the "T" in string w with the "G" in string v 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. Using only one matrix to backtrack is not sufficient for every case.

Input:

```
5 2 15 5
ACGTA
ACT
```

Output:

```
-12
ACGTA
ACT--
```

Case 10

Description: A larger dataset of the same size as that provided by the randomized autograder. Check input/output folders for this dataset.