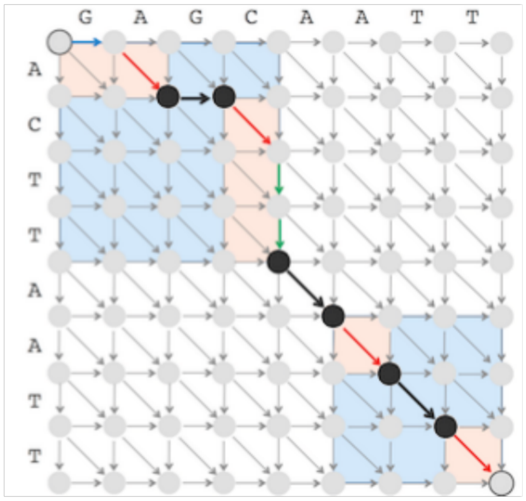
5L Align Two Strings Using Linear Space

Global Alignment in Linear Space Problem
Find a highest-scoring global alignment between two strings using a scoring matrix in linear space.

Input: Two amino acid strings, a 20×20 scoring matrix, and an indel penalty.
Output: A highest-scoring global alignment of these two strings using linear space (with respect to the scoring parameters) and its score.



,

Formatting

Input: Three space-separated integers representing the match reward, mismatch penalty, and indel penalty respectively, followed by two newline-separated DNA strings v and w.
Output: The maximum global alignment score of v and w as an integer as determined by the scoring function followed by a newline-separated global alignment of v and w achieving this maximum score (if multiple global alignments achieving the maximum score exist, you may return any one).

Constraints

- The lengths of v and w will be between 1 and 10^5.
- Both v and w will be amino acid strings.
- All parameters of the scoring function will be between 1 and 10^1.

Test Cases

Case 1

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

Input:
```
1 1 2
GAGA
GAT
```

Output:
```
-1
GAGA
GA-T
```

Case 2

Description: The majority of bugs for this problem will likely be due to some uncaught mistake in the implementation of the Finding a Middle Edge in Alignment Graph in Linear Space Problem. Be sure that your middle edge implementation passes all given tests before working on this problem. This test makes sure that your code can handle runs of indels in the reconstructed alignment. If your score is incorrect be sure to check to make sure you are assigning the correct score to the middle edge. The error may stem from misidentification of the type of middle edge (match, mismatch, or indel) or from an indexing error when comparing the characters of the two strings to distinguish between matches and mismatches. If your output contains any mismatches there is probably some error in your reconstruction of the alignment.

Input:
```
1 5 1
TT
CC
```

Output:
```
-4
--TT
CC--
```

Case 3

Description: This test makes sure that your code correctly mismatches characters when the ideal alignment requires it. In test dataset 1, your code was tested for its ability to assign continuous indels when necessary; this dataset tests if your code is able to assign continuous mismatches if necessary. If there are any indels in your alignment and your score is correct, double check that your reconstruction methods are correct. If your score is incorrect check that your base cases correct update the score. Pay especial attention to the base cases of the recursive algorithm in both the score updating and alignment reconstruction steps.

Input:
```
1 1 5
TT
CC
```

Output:
```
-2
TT
CC
```

Case 4

Description: This test makes sure that your code correctly aligns the upper and lower sub-matrices after recursive calls. Be very careful to check that your *top*, *bottom*, *left*, and *right* values are correct for each recursive call since it's very easy to have an off-by-one error that will become very difficult to debug on large datasets. To help debug any errors you may have on this dataset a trace of *top*, *bottom*, *left*, and *right* values is provided below. Your code does not necessarily need to have the exact same values as below (if there are ties in *Length* scores).

Input:
```
1 5 1
GAACGATTG
GGG
```

Output:
```
-3
GAACGATTG
G---G---G
```

Case 5

Description: This test makes sure that your code correctly handles inputs in which the match score is not equal to one. If your output doesn't match the correct output make sure that your implementation doesn't make any assumptions about the scoring scheme of the dataset. It is possible that your code passes all previous datasets and fails this one while assuming the match score is equal to one. Make sure that your implementation uses the match score given in the input instead of hard-coding any value for the match score. This requirement applies to the mismatch and indel penalties as well, but those elements of the scoring scheme have varied in previous datasets and would likely cause an earlier test failure.

Input:

```
2 3 1
GCG
CT
```

Output:

```
-1
GCG-
-C-T
```

Case 6

Description: This test makes sure that your code correctly handles inputs in which string *w* is one character long. The correctness of your output for this dataset is largely reliant on the correctness of your underlying middle edge implementation. If your output doesn't match the correct output make sure that your implementation of the middle edge algorithm passes test dataset 5 in the Finding a Middle Edge in Alignment Graph in Linear Space Problem. That test also considers the case where string *w* is one character long.

Input:

```
1 2 3
ACAGCTA
G
```

Output:

```
-17
ACAGCTA
---G---
```

Case 7

Description: This test makes sure that your code correctly handles inputs in which string v is one character long. The correctness of your output for this dataset is largely reliant on the correctness of your underlying middle edge implementation. If your output doesn't match the correct output make sure that your implementation of the middle edge algorithm passes test dataset 6 in the Finding a Middle Edge in Alignment Graph in Linear Space Problem. That test also considers the case where string v is one character long.

Input:
```
3 4 1
A
CGGAGTGCC
```

Output:
```
-5
---A-----
CGGAGTGCC
```

Case 8

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