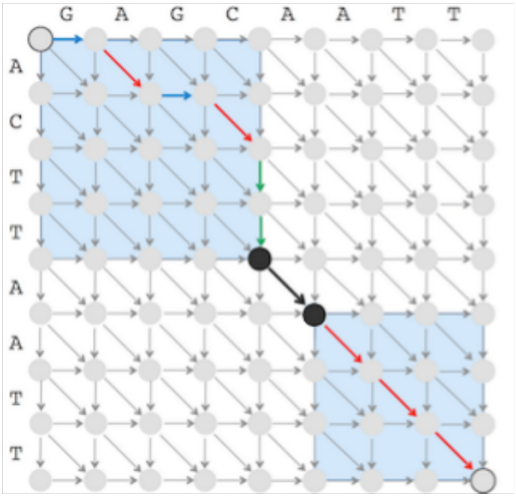
5K Find a Middle Edge in an Alignment Graph in Linear Space

Middle Edge in Linear Space Problem
Find the middle edge in an alignment graph in linear space.

Input: Two amino acid strings.
Output: A middle edge in the alignment graph of these two strings.



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: A newline-separated pair of nodes as space-separated x and y coordinate pairs, where the first node is connected to the second node through the middle edge of the matrix created using the scoring function.

Constraints

- The lengths of v and w will be between 1 and 10^2
- 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:

```
2 2
2 3
```

Case 2

Description: This test makes sure that your code can identify horizontal middle edges. In the sample dataset the middle edge is diagonal, but the high mismatch penalty in this dataset forces indels into the alignment. The middle node can be either (0, 2) or (1, 2) but either way the middle edge must be horizontal ((1, 2) and (1, 3) is also a valid answer). If your middle node is incorrect make sure that you're combining *FromSource* and *ToSink* correctly. If the middle node is correct but your middle edge is incorrect it's likely that you're making some indexing mistake in choosing the second node for the edge. If you're reversing the strings to calculate *ToSink* remember to compensate for that reversal when choosing the indices for the second node in the middle edge.

Input:

```
1 5 1
TTTT
CC
```

Output:

```
0 2
0 3
```

Case 3

Description: This test makes sure that your code can handle finding the middle edge when the first string has an odd length. The definition of the middle column is $\lfloor \frac{m}{2} \rfloor$ where m is the length of the string along the horizontal axis. In this case it should be the second column, indexed by 1. If your middle node isn't (0, 1) then check to make sure you're actually using the correct middle column for a string of odd length.

Input:

```
1 1 2
GAT
AT
```

Output:

```
0 1
1 2
```

Case 4

Description: This test makes sure that your code can identify vertical middle edges. In the sample dataset the middle edge is diagonal, but the structure of the strings in this dataset makes mismatches detrimental. There are two possible middle nodes for this dataset: (2, 2) or (3, 2). The middle edge starting at (2, 2) is vertical while the middle edge starting at (3, 2) is diagonal. If you want to make sure that your code can identify vertical middle edges choose the first maximum score in *Length*. In this dataset, *Length* should be $[-3, 0, 3, 3, 0, -3]$. If you choose the first 3 as the location of the middle node you'll have a vertical middle edge corresponding to the first possible answer. If you choose the second 3 you will have a diagonal middle edge corresponding to the second possible answer. Both answers are correct but it may be valuable in your debugging to test if your code can identify vertical edges when they're a possibility.

Input:

```
1 1 1
TTTT
TTCTT
```

Output:

```
2 2
3 2
```

Case 5

Description: This test makes sure that your code correctly identifies the middle edge when the maximum value in *Length* is the last value. This means that the middle edge must be horizontal. Depending on how you're keeping track of previous values for the *ToSink* array it is easy to introduce bugs when the middle node falls on the first or last value in *ToSink*. If your code correctly finds the middle node but has the incorrect middle edge (i.e. the second node in your output is incorrect) then you are likely making a mistake in checking the previous values for the *ToSink* array. Double check to make sure your determination of previous values is valid for edge cases in the *ToSink* array.

Input:

```
1 5 1
GAACCC
G
```

Output:

```
1 3
1 4
```

Case 6

Description: This test makes sure that your code correctly handle 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
ACAGT
CAT
```

Output:

```
1 2
2 3
```

Case 7

Description: This test makes sure that your code correctly handle inputs where the length of string v is equal to one. This dataset is similar to test dataset 5 except string v is one character long instead of string w. There are multiple possible middle edges for this dataset. If your output doesn't match one of the correct outputs make sure that your implementation explicitly considers a case in which string v is only one character long. Double check that your middle column is equal to zero. Also make sure that your middle column being equal to zero doesn't invalidate any part of your implementation. Note that (1, 0) (2, 0) and (2, 0) (3, 1) are also valid answers.

Input:

```
2 5 3
T
AATCCC
```

Output:

```
0 0
1 0
```

Case 8

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