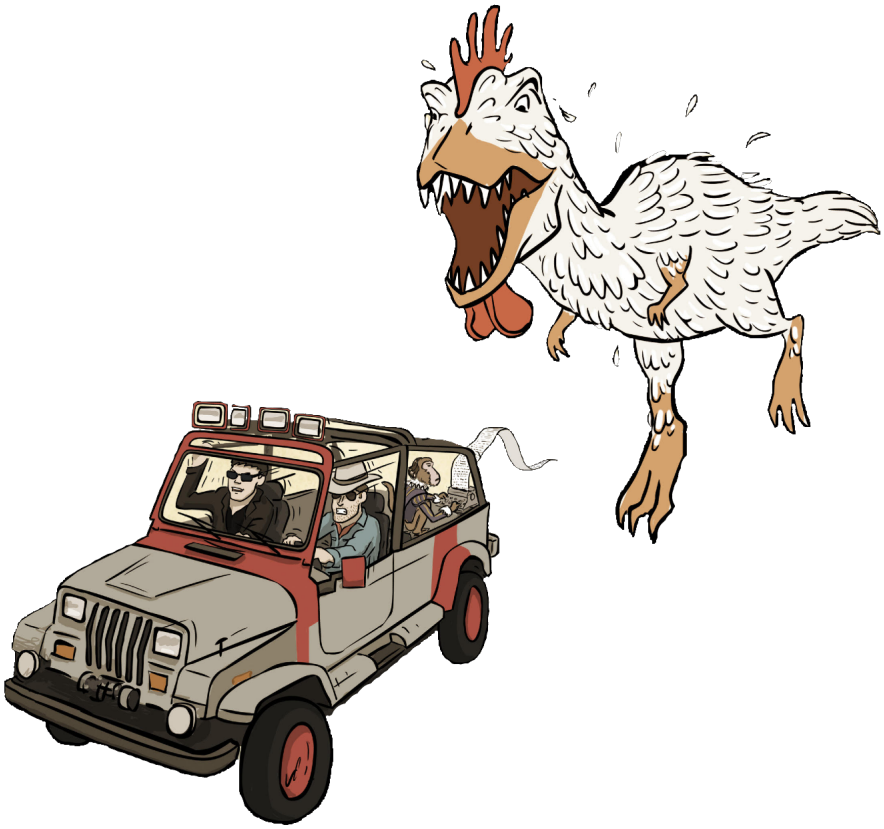
Problem Register for
Bioinformatics Algorithms: An Active Learning Approach

Parker Côté and Ryan Eveloff



Meet the Authors

PARKER CÔTÉ graduated from the Department of Biological Sciences at the University of California, San Diego in 2021 and currently works in software development at Abterra Biosciences, an antibody discovery and sequencing company based in San Diego. His research interests include precision medicine, degenerative disease, comparative genomics, and proteomics. Aside from his work in the field, he enjoys cycling, cooking, and playing music.

RYAN EVELOFF is a graduating senior at the University of California, San Diego in the Computer Science and Engineering department. He is particularly interested in precision medicine, neurogenetic disease, and data-informed healthcare. In 2021, he completed a joint immunology fellowship between UC San Diego and the J. Craig Venter Institute. Outside of his work, Ryan enjoys hiking around San Diego, watching college basketball, and playing electric bass.

Why We Made This

After completing Dr. Pevzner's Bioinformatics Algorithms course at the University of California, San Diego during the Winter 2021 quarter, we came to the conclusion that a centralized, syntactically consistent register of all problems, including reproducible test cases, would have been an invaluable resource to us and our classmates. Namely, we often found ourselves stuck trying to find an edge case or confused about the parameters of a problem, but no resource existed to satisfy that need. With this problem register, we aim to fill that need.

Our immense thanks goes out to Dr. Pevzner and Dr. Compeau for giving us the opportunity to pursue this project and for offering creative directions, as well as Andrey Bzikadze and Vikram Sirupurapu for assisting in the brainstorming and proofreading processes. We hope that you can utilize this register as a resource to enrich your own learning and that of those around you.

Best,
Parker & Ryan

Prologue

Things to know before solving Programming Challenges

Unless specified otherwise, the following are *soft* rules and will not count for grading, they simply lay out the format for *our* test case output.

- Space separation for list elements
- Inputs and outputs are case sensitive
- When outputting integers, do not include the floating point (E.g. 1 instead of 1.0)
- Lexicographical ordering

Unless specified otherwise, the following are *hard* rules and *will* count for grading.

- Zero-indexing

Good	Bad
`[0, 1, 2, 3][0] = 0`	`[0, 1, 2, 3][1] = 0`

- Space separation for list elements

Good	Bad
`0 1 2 3`	`0, 1, 2, 3`

- Newline separation for arguments

Good	Bad
`1` `2`	`1, 2`

Test Cases

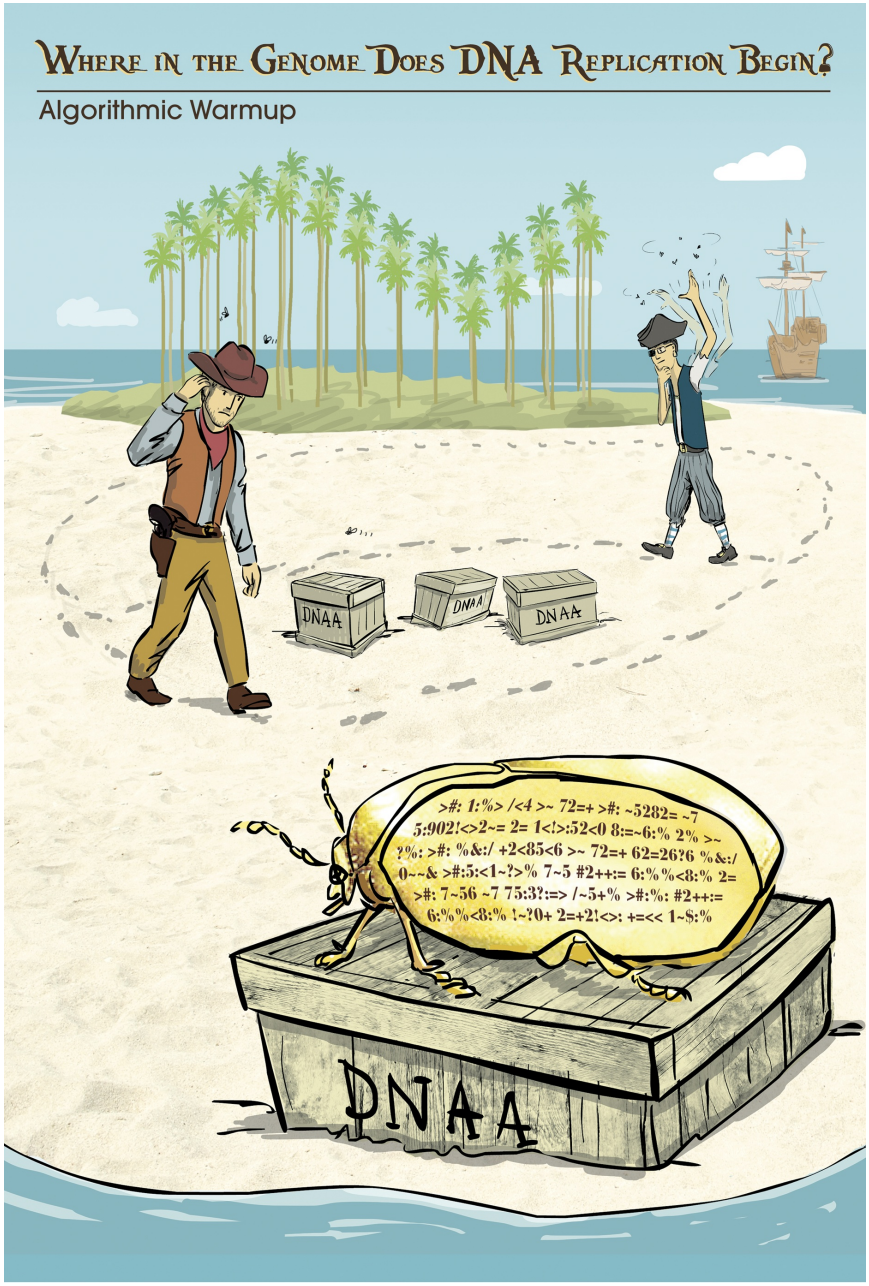
While we provide a range of test cases for each problems, these are *not* exhaustive. We encourage you to investigate each problem to determine the potential edge cases and relevant parameters, then write your own tests as you see fit. Similarly, many of our test cases are based on the DNA alphabet (A, C, G, and T). When designing your algorithms, unless explicitly stated otherwise, you should be prepared to test on all characters, not just the characters in the genetic alphabet. Please note that the constraints provided for each question will not change and you will not be tested on datasets outside of those parameters. You should consider the effects of these constraints on run-time and memory when designing your algorithmic solutions. Lastly, you may see ellipses (...) in test case input/output. These are *not* part of the test case and only illustrate a continuation from the previous line.

All test cases are also available on the GitHub associated with this Problem Register. Keep an eye out for symbols that link each problem to a folder of its test cases.

Contents

1 Where in the Genome Does DNA Replication Begin?

Algorithmic Warmup



1A Compute the Number of Occurrences of a Pattern in a Text

Pattern Count Problem
Implement PatternCount.

Input: Strings *Text* and *Pattern*.
Output: The number of occurrences of *Pattern* in *Text*.

```
AGAGATCAGA
AGAGA   AGA
1  2    3
```

Formatting

Input: Newline-separated strings *Text* and *Pattern*.
Output: An integer representing the number of times *Pattern* appears in *Text*.

Constraints

- The length of *Text* will be between 1 and 10^4.
- The length of *Pattern* will be between 1 and 10^1.
- *Text* and *Pattern* will be DNA strings.

Test Cases

Case 1

Description: The sample dataset is not actually run on your code. Notice that GCG occurs twice in *Text*: once at the beginning (**GCG**CG) and once at the end (GC**GCG**). A common mistake for this problem is incorrectly handling overlaps and not counting the second of these two occurrences (because it begins at the end of the previous occurrence).

Input:

```
GCGCG
GCG
```

Output:

```
2
```

Case 2

Description: This dataset just checks if you're correctly counting. It is the "easiest" test. Notice that all occurrences of CG in *Text* (ACGTACGTACGT) are away from the very edges (so your code won't fail on off-by-one errors at the beginning or at the end of *Text*) and that none of the occurrences of Pattern overlap (so your code won't fail if you fail to account for overlaps).

Input:

```
ACGTACGTACGT
CG
```

Output:

```
3
```

Case 3

Description: This dataset checks if your code correctly handles cases where there is an occurrence of *Pattern* at the very beginning of *Text*. Note that there are no overlapping occurrences of *Pattern* (i.e. AAAA), and there is no occurrence of Pattern at the very end of *Text*, so assuming your code passed Test Dataset 1, this test would only check for off-by-one errors at the beginning of *Text*.

Input:

```
AAAGAGTGTCTGATAGCAGCTTCTGAACTGGTTACCTGCCGTGAGTAAATTAAATTTTATTGACTTAGG...
...TCACTAAATACTTTAACCAATATAGGCATAGCGCACAGACAATAATAATTACAGAGTACACAACATCCA
AAA
```

Output:

```
3
```

Case 4

Description: This dataset checks if your code correctly handles cases where there is an occurrence of *Pattern* at the very end of *Text*. Note that there are no overlapping occurrences of *Pattern* (i.e. AAAA), and there is no occurrence of *Pattern* at the very beginning of *Text*, so assuming your code passed Test Dataset 2, this test would only check for off-by-one errors at the end of *Text*.

Input:

```
AGCGTGCCGAAATATGCCGCCAGACCTGCTGCGGTGGCCTCGCCGACTTCACGGATGCCAAGTGCATAG...
...AGGAAGCGAGCAAAGGTGGTTTCTTTCGCTTTATCCAGCGCTTAACCACGTTCTGTGCCGACTTT
TTT
```

Output:

```
4
```

Case 5

Description: This test dataset checks if your code is also counting occurrences of the Reverse Complement of *Pattern* (which would have an output of 4), which is out of the scope of this problem (that will come up later in the chapter). Your code should only be looking for perfect matches of *Pattern* in *Text* at this point.

Input:

```
GGACTTACTGACGTACG
ACT
```

Output:

```
2
```

Case 6

Description: This dataset checks if your code correctly handles cases where occurrences of *Pattern* overlap. For example, any occurrence of the string CCC should count as 2 occurrences of CC (**CC**C and C**CC**). In this dataset, there are 5 occurrences of CC including overlaps. (AT**CC**GAT**CCC**ATG**CCC**ATG)

Input:

```
ATCCGATCCCATGCCCATG
CC
```

Output:

```
5
```

Case 7

Description: This is the final test that we run your code on: the full dataset.

Input:

```
CTGTTTTTGATCCATGATATGTTATCTCTCCGTCATCAGAAGAACAGTGACGGATCGCCCTCTCTCTTG...
...GTCAGGCGACCGTTTGCCATAATGCCCATGCTTTCCAGCCAGCTCTCAAACTCCGGTGACTCGCGC...
...AGGTTGAGT
CTC
```

Output:

```
9
```

Case 8

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

1B Find the Most Frequent Words in a String

Frequent Words Problem
Find the most frequent words k-mers in a string.

Input: A DNA string *Text* and an integer *k*.
Output: All most frequent *k*-mers in *Text* (in any order).

```
AGAGACGTGAGAG
AGAGA     AGA
 GAG     GAGAG
```

Formatting

Input: A DNA string *Text* followed by an integer *k*.
Output: All most frequent *k*-mers in *Text* (in any order).

Constraints

- The length of *Text* will be between 1 and 10^4.
- The integer *k* will be between 1 and 10^2.
- *Text* will be a DNA string.

Test Cases

Case 1

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

Input:

```
ACGTTGCATGTCGCATGATGCATGAGAGCT
4
```

Output:

```
CATG GCAT
```

Case 2

Description: This dataset just checks if you're counting the first *k*-mer in *Text* (`TGG` in this example). If you do not count the first *k*-mer (`TGG`), you will get the following "most frequent" *k*-mers in addition to `TGG`: `ACT`, `CAC`, `CCA`, `CTT`, `GGT`.

Input:

```
TGGTAGCGACGTTGGTCCCGCCGCTTGAGAATCTGGATGAACATAAGCTCCCACTTGGCTTATTCAGAG...
...AACTGGTCAACACTTGTCTCTCCCAGCCAGGTCTGACCACCGGGCAACTTTTAGAGCACTATCGTG...
...GTACAAATAATGCTGCCAC
3
```

Output:

```
TGG
```

Case 3

Description: This dataset just checks if you're counting the last *k*-mer in *Text* (`TTTT` in this example). If you do not count the last *k*-mer (`TTTT`), you will get the following "most frequent" *k*-mers in addition to `TTTT`: `AACG`, `AATA`, `ACAA`, `CAAC`, `CTGG`, `CTGG`, `CTTT`, `TTGC`, `TTTG`.

Input:

```
CAGTGGCAGATGACATTTTGCTGGTCGACTGGTTACAACAACGCCTGGGGCTTTTGAGCAACGAGACTTT...
...TCAATGTTGCACCGTTTGCTGCATGATATTGAAAACAATATCACCAAATAAATAACGCCTTAGTAAG...
...TAGCTTTT
4
```

Output:

```
TTTT
```

Case 4

Description: This dataset checks if your code correctly handles cases where there are overlapping occurrences of *Pattern* throughout *Text*. For example, AACAACAA contains two occurrences of AACAA (**AACAA**CAA and AAC**AACAA**), so if your code counts AACAACAA as one occurrence of AACAA, your code will fail on this test case.

Input:

```
ATACAATTACAGTCTGGAACCGGATGAACTGGCCGCAGGTTAACAACAGAGTTGCCAGGCACTGCCGCTG...
...ACCAGCAACAACAACAATGACTTTGACGCGAAGGGGATGGCATGAGCGAACTGATCGTCAGCCGTCA...
...GCAACGAGTATTGTTGCTGACCCTTAACAATCCCGCCGCACGTAATGCGCTAACTAATGCCCTGCTG
5
```

Output:

```
AACAA
```

Case 5

Description: This test dataset checks if your code correctly handles ties (i.e. your code actually outputs ALL "most frequent" *k*-mers, and not just a single "most frequent" *k*-mer). For example, in the string ATATA, there are two "most frequent" *k*-mers: AT and TA. AT occurs twice (**ATAT**A), and TA occurs twice (A**TATA**), so both of these should be output (separated by a space character).

Input:

```
CCAGCGGGGGTTGATGCTCTGGGGGTCACAAGATTGCATTTTTATGGGGTTGCAAAAATGTTTTTTACGG...
...CAGATTCATTTAAAATGCCCACTGGCTGGAGACATAGCCCGGATGCGCGTCTTTTACAACGTATTGC...
...GGGGTAAAATCGTAGATGTTTTAAAATAGGCGTAAC
5
```

Output:

```
AAAAT GGGGT TTTTA
```

Case 6

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

1C Find the Reverse Complement of a String

Reverse Complement Problem
Find the reverse complement of a DNA string.

Input: A DNA string *Pattern*.
Output: The reverse complement of the string *Pattern*.

Formatting

Input: A DNA string *Pattern*.
Output: A string representing $\overline{Pattern}$, the reverse complement of *Pattern*.

Constraints

- The length of *Pattern* will be between 1 and 10^4.
- *Pattern* will be a DNA string.

Test Cases

Case 1

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

Input:
`AAAACCCGGT`

Output:
`ACCGGGTTTT`

Case 2

Description: This test dataset performs two checks. First, it makes sure that you didn't forget to complement *Pattern* (i.e. you reversed *Pattern*, but not complemented), which would have an output of `CACACA`. Then, it makes sure that you didn't forget to reverse *Pattern* (i.e. you complemented *Pattern*, but not reversed), which would have an output of `TGTGTG`.

Input:
`ACACAC`

Output:
`GTGTGT`

Case 3

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

1D Find All Occurrences of a Pattern in a String

Pattern Matching Problem
Find all occurrences of a Pattern in a string.

Input: DNA strings *Pattern* and *Genome*.
Output: All starting positions in *Genome* where *Pattern* appears as a substring.

```
AGAGATCAGA
AGAGA   AGA
0 2     7
```

Formatting

Input: DNA strings *Pattern* and *Genome*.
Output: A space-separated list of integers representing each starting position in *Genome* where *Pattern* appears as a substring.

Constraints

- The length of *Pattern* will be between 1 and 10^1.
- The length of *Genome* will be between 1 and 10^4.
- *Pattern* and *Genome* will be DNA strings.

Test Cases

Case 1

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

Input:

```
ATAT
GATATATGCATATACTT
```

Output:

```
1 3 9
```

Case 2

Description: This dataset checks if your code is written correctly but is also taking into account reverse complements, which we are not yet doing. Even though the reverse complement of `ACAC` (which is `GTGT`) occurs in *Genome*, we only want to count occurrences of `ACAC` specifically, which only occurs at index `4`.

Input:

```
ACAC
TTTTACACTTTTTTGTGTAAAAA
```

Output:

```
4
```

Case 3

Description: This dataset checks for off-by-one errors at the beginning of *Genome*. Notice that `AAA` occurs at the very beginning of *Genome*, so if you were to miss the first *k*-mer of *Genome*, your code would output the following: `46 51 74`.

Input:

```
AAA
AAAGAGTGTCTGATAGCAGCTTCTGAACTGGTTACCTGCCGTGAGTAAATTAAATTTTATTGACTTAGGT...
...CACTAAATACTTTAACCAATATAGGCATAGCGCACAGACAGATAATAATTACAGAGTACACAACATCCAT
```

Output:

```
0 46 51 74
```

Case 4

Description: This dataset checks for off-by-one errors at the end of *Genome*. Notice that `TTT` occurs at the very end of *Genome*, so if you were to miss the last *k*-mer of *Genome*, your code would output the following: `88 93 98`.

Input:

```
TTT
AGCGTGCCGAAATATGCCGCCAGACCTGCTGCGGTGGCCTCGCCGACTTCACGGATGCCAAGTGCATAGA...
...GGAAGCGAGCAAAGGTGGTTTCTTTCGCTTTATCCAGCGCGTTAACCACGTTCTGTGCCGACTTT
```

Output:

```
88 92 98 132
```

Case 5

Description: This test dataset checks if your code correctly handles cases where instances of *Pattern* overlap in *Genome*. In this case, if you did not count overlaps, you would only find the first and last instances of `ATA` (**`ATA`**`TATA` and `ATAT`**`ATA`**). However, there is indeed a third occurrence, where the other two overlap (`AT`**`ATA`**`TA`).

Input:

```
ATA
ATATATA
```

Output:

```
0 2 4
```

Case 6

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

1E Find Patterns Forming Clumps in a String

Clump Finding Problem
Find patterns forming clumps in a string.

Input: A DNA string *Genome*, and integers k, L, and t.
Output: All distinct k-mers forming (L, t)-clumps in *Genome*.

aggtccTGCAATGCATGAAGCCTGCAtgtt

Formatting

Input: A DNA string *Genome* followed by space-separated integers k, L, and t.
Output: A space-separated list of strings containing all distinct k-mers forming (L, t)-clumps in *Genome*.

Constraints

- The length of *Genome* will be between 1 and 10^4.
- The integer k will be between 1 and 10^1.
- The integer L will be between 1 and 10^3.
- The integer t will be between 1 and 10^2.
- *Genome* will be a DNA string.

Test Cases

Case 1

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

Input:

```
CGGACTCGACAGATGTGAAGAAATGTGAAGACTGAGTGAAGAGAAGAGGAAACACGACACGACATTGCGA...
...CATAATGTACGAATGTAATGTGCCTATGGC
5 75 4
```

Output:

```
CGACA GAAGA AATGT
```

Case 2

Description: This dataset makes sure that your code only counts k-mers that fall *completely* within a given L-window. For example, take the 4-window starting at index 4 (`AAAACGTCGAAAAA`). One might think that the 2-mer `CG` occurs twice in this window since the first letter of the second occurrence happens at the very end of the window. However, since the second occurrence of `CG` does not fall entirely in this 4-window, it does not count. Thus, the only result is `AA`.

Input:

```
AAAACGTCGAAAAA
2 4 2
```

Output:

```
AA
```

Case 3

Description: This dataset checks if your code has an off-by-one error when checking k-mers within an L-window. Notice that, for each 1-mer (`A`, `C`, `G`, and `T`), there are 3 nucleotides between the first and second occurrence. In other words, each nucleotide occurs twice in a specific 5-window: once at the beginning of the 5-window, and once at the end: **A**CGT**A**CGT A**C**GTA**C**GT AC**G**TAC**G**T ACG**T**ACG**T**.

Input:

```
ACGTACGT
1 5 2
```

Output:

```
A C G T
```

Case 4

Description: This dataset checks if your code is correctly handling overlapping *k*-mers. For example, ATA forms a (5, 2)-clump in CCCATATACCC (CCC**ATATA**CCC and CCCAT**ATACC**C).

Input:

```
CCACGCGGTGTACGCTGCAAAAAGCCTTGCTGAATCAAATAAGGTTCCAGCACATCCTCAATGGTTTCAC...
...GTTCTTCGCCAATGGCTGCCGCCAGGTTATCCAGACCTACAGGTCCACCAAAGAACTTATCGATTAC...
...CGCCAGCAACAATTTGCGGTCCATATAATCGAAACCTTCAGCATCGACATTCAACATATCCAGCG
3 25 3
```

Output:

```
AAA CAG CAT GCC TTC
```

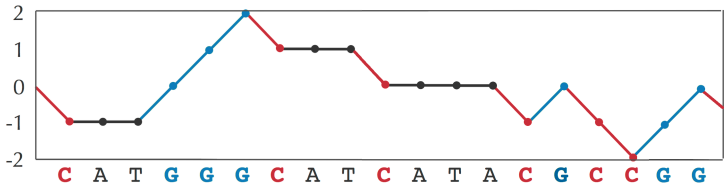
Case 5

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

1F Find a Position in a Genome Minimizing the Skew

Minimum Skew Problem
Find a position in a genome minimizing the skew.

Input: A DNA string *Genome*.
Output: All integers i minimizing SKEW(PREFIX$_i$(*Genome*)) over all values of i (from 0 to |*Genome*|).



Formatting

Input: A DNA string *Genome*.
Output: A space-separated list of integers i minimizing SKEW(PREFIX$_i$(*Genome*) over all values of i (from 0 to |*Genome*|).

Constraints

- The length of *Genome* will be between 1 and 10^5.
- *Genome* will be a DNA string.

Test Cases

Case 1

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

Input:
```
TAAAGACTGCCGAGAGGCCAACACGAGTGCTAGAACGAGGGGCGTAAACGCGGGTCCGAT
```

Output:
```
11 24
```

Case 2

Description: This dataset checks if your code's indexing is off. Specifically, it verifies that your code is not returning an index 1 too high (i.e. 4) or 1 too low (i.e. 2).

Input:
```
ACCG
```

Output:
```
3
```

Case 3

Description: This dataset checks to see if your code is missing the last symbol of *Genome*.

Input:
```
ACCC
```

Output:
```
4
```

Case 4

Description: This dataset makes sure you're not accidentally finding the maximum skew instead of the minimum skew.

Input:
```
CCGGGT
```

Output:
```
2
```

Case 5

Description: First, this dataset checks if you are only finding 1 index (and not multiple indices). Then, it checks if you are using a delimiter to separate your indices (a space character).

Input:

```
CCGGCCGG
```

Output:

```
2 6
```

Case 6

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

1G Compute the Hamming Distance Between Two Strings

Hamming Distance Problem
Compute the Hamming distance between two strings.

Input: Two strings of equal length.
Output: The Hamming distance between these strings.

Formatting

Input: Two DNA strings $Text_1$ and $Text_2$.
Output: An integer representing the Hamming distance between $Text_1$ and $Text_2$.

Constraints

- The length of $Text_1$ and $Text_2$ will be between 1 and 10^4.
- $Text_1$ and $Text_2$ will have equal lengths.
- $Text_1$ and $Text_2$ will be DNA strings.

Test Cases

Case 1

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

Input:

```
GGGCCGTTGGT
GGACCGTTGAC
```

Output:

```
3
```

Case 2

Description: This dataset checks if your code isn't keeping count (i.e. returns 0 when the answer is clearly nonzero) or if your code returns a negative value, which is impossible.

Input:

```
AAAA
TTTT
```

Output:

```
4
```

Case 3

Description: This dataset checks if your code is finding Edit Distance (which would be 2) instead of Hamming Distance.

Input:

```
ACGTACGT
TACGTACG
```

Output:

```
8
```

Case 4

Description: This dataset checks if your code is returning the number of matches (2) instead of the number of mismatches (6).

Input:

```
ACGTACGT
CCCCCCCC
```

Output:

```
6
```

Case 5

Description: This dataset checks if your code works on a dataset where the two input strings have no matches.

Input:

```
ACGTACGT
TGCATGCA
```

Output:

```
8
```

Case 6

Description: This dataset checks if you have an off-by-one error at the beginning (i.e. you are starting at the second character of the strings instead of the first character).

Input:

```
GATAGCAGCTTCTGAACTGGTTACCTGCCGTGAGTAAATTAAAATTTTATTGACTTAGGTCACTAAATACT
AATAGCAGCTTCTCAACTGGTTACCTCGTATGAGTAAATTAGGTCATTATTGACTCAGGTCACTAACGTCT
```

Output:

```
15
```

Case 7

Description: This dataset checks if you have an off-by-one error at the end (i.e. you are ending at the second-to-last character of the strings instead of the last character).

Input:

```
GATAGCAGCTTCTGAACTGGTTACCTGCCGTGAGTAAATTAAAATTTTATTGACTTAGGTCACTAAATACT
AATAGCAGCTTCTCAACTGGTTACCTCGTATGAGTAAATTAGGTCATTATTGACTCAGGTCACTAACGTCT
```

Output:

```
28
```

Case 8

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

1H Find All Approximate Occurrences of a Pattern in a String

Approximate Pattern Matching Problem
Find all approximate occurrences of a pattern in a string.

Input: DNA strings *Pattern* and *Text* along with an integer d.
Output: All starting positions where *Pattern* appears as a substring of *Text* with at most d mismatches.

```
CGACTAGTTT
CGACGA
0  3
```

Formatting

Input: DNA strings *Pattern* and *Text* along with an integer d.
Output: A space-separated list of integers containing all starting positions where *Pattern* appears as a substring of *Text* with at most d mismatches.

Constraints

- The length of *Pattern* will be between 1 and 10^2.
- The length of *Text* will be between 1 and 10^5.
- The integer d will be between 1 and 10^1.
- *Pattern* and *Text* will be DNA strings.

Test Cases

Case 1

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

Input:

```
ATTCTGGA
CGCCCGAATCCAGAACGCATTCCCATATTTCGGGACCACTGGCCTCCACGGTACGGACGTCAATCAAAT
3
```

Output:

```
6 7 26 27
```

Case 2

Description: This dataset checks if you are only counting instances where the number of mismatches is exactly equal to d (i.e. ignoring instances where mismatch < d).

Input:

```
AAA
TTTTTTAAATTTTAAATTTTTT
2
```

Output:

```
4 5 6 7 8 11 12 13 14 15
```

Case 3

Description: This dataset checks if your code has an off-by-one error at the beginning of *Text* (i.e. your code is not checking the the left-most substring of *Text*).

Input:

```
GAGCGCTGG
GAGCGCTGGGTTAACTCGCTACTTCCCGACGAGCGCTGTGGCGCAAATTGGCGATGAAACTGCAGAGAGAACTG...
...GTCATCCAACTGAATTCTCCCCGCTATCGCATTTTGATGCGCGCCGCGTCGATT
2
```

Output:

```
0 30 66
```

Case 4

Description: This dataset checks if your code has an off-by-one error at the end of *Text* (i.e. your code is not checking the the right-most substring of *Text*).

Input:

```
AATCCTTTCA
CCAAATCCCCTCATGGCATGCATTCCCGCAGTATTTAATCCTTTCATTCTGCATATAAGTAGTGAAGGT...
...ATAGAAACCCGTTCAAGCCCGCAGCGGTAAAACCGAGAACCATGATGAATGCACGGCGATTGCGCC...
...ATAATCCAAACA
3
```

Output:

```
3 36 74 137
```

Case 5

Description: This dataset checks if your code is correctly accounting for overlapping instances of *Pattern* in *Text*.

Input:

```
CCGTCATCC
CCGTCATCCGTCATCCTCGCCACGTTGGCATGCATTCCGTCATCCCGTCAGGCATACTTCTGCATATAA...
...GTACAAACATCCGTCATGTCAAAGGGAGCCCGCAGCGGTAAAACCGAGAACCATGATGAATGCACG...
...GCGATTGC
3
```

Output:

```
0 7 36 44 48 72 79 112
```

Case 6

Description: This dataset checks if you are only counting instances of *Pattern* with less than *d* mismatches (as opposed to instances of *Pattern* with less than or equal to *d* mismatches).

Input:

```
TTT
AAAAAA
3
```

Output:

```
0 1 2 3
```

Case 7

Description: This dataset checks if your code works with input where $d = 0$ (i.e. only perfect matches are allowed).

Input:

```
CCA
CCACCT
0
```

Output:

```
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.

1I Find the Most Frequent Words with Mismatches in a String

Frequent Words with Mismatches Problem
Find the most frequent k-mers with mismatches in a string.

Input: A DNA string *Text* as well as integers k and d.
Output: All most frequent k-mers with up to d mismatches in *Text*.

```
CGACTAGTTT
 ATT  ATT
   ATT
```

Formatting

Input: A DNA string *Text* as well as integers k and d.
Output: A space-separated list of strings representing all most frequent k-mers with up to d mismatches in *Text*.

Constraints

- The length of *Text* will be between 1 and 10^3.
- The integer k will be between 1 and 10^1.
- The integer d will be between 1 and 10^1.
- *Text* will be a DNA string.

Test Cases

Case 1

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

Input:

```
ACGTTGCATGTCGCATGATGCATGAGAGCT
4 1
```

Output:

```
ATGC ATGT GATG
```

Case 2

Description: *Text* contains partial and complete matches for the most frequent word.

Input:

```
AGGT
2 1
```

Output:

```
GG
```

Case 3

Description: $d = 0$.

Input:

```
AGGGT
2 0
```

Output:

```
GG
```

Case 4

Description: *Text* has multiple most frequent words.

Input:

```
AGGCGG
3 0
```

Output:

```
AGG GGC GCG CGG
```

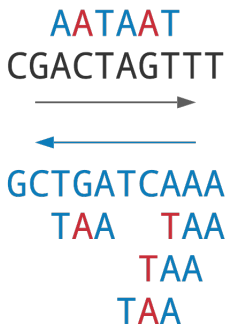
Case 5

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

1J Find Frequent Words with Mismatches and Reverse Complements

Frequent Words with Mismatches and Reverse Complements Problem
Find the most frequent k-mers (with mismatches and reverse complements) in a DNA string.

Input: A DNA string *Text* as well as integers k and d.
Output: All k-mers *Pattern* maximizing the sum $\text{COUNT}_d(Text, Pattern)+\text{COUNT}_d(Text, \overline{Pattern})$ over all possible k-mers.



Formatting

Input: A DNA string *Text* as well as integers k and d.
Output: A space-separated list of strings representing all k-mers *Pattern* maximizing the sum $\text{COUNT}_d(Text, Pattern)+\text{COUNT}_d(Text, \overline{Pattern})$ over all possible k-mers.

Constraints

- The length of *Text* will be between 1 and 10^3.
- The integer k will be between 1 and 10^1.
- The integer d will be between 1 and 10^1.
- *Text* will be a DNA string.

Test Cases

Case 1

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

Input:
```
ACGTTGCATGTCGCATGATGCATGAGAGCT
4 1
```

Output:
```
ACAT ATGT
```

Case 2

Description: *Text* contains partial and completes matches for the most frequent word.

Input:
```
AAAAAAAAAA
2 1
```

Output:
```
AT TA
```

Case 3

Description: This dataset makes sure that your code is not accidentally swapping k and d.

Input:
```
AGTCAGTC
4 2
```

Output:
```
AATT GGCC
```

Case 4

Description: This dataset makes sure you are finding k-mers in both *Text* and the reverse complement of *Text*.

Input:
```
AATTAATTGGTAGGTAGGTA
4 0
```

Output:
```
AATT
```

Case 5

Description: This dataset first checks that k-mers with exactly d mismatches are being found. Then, it checks that k-mers with less than d mismatches are being allowed (i.e. you are not only allowing k-mers with exactly d mismatches). Next, it checks that you are not returning too few k-mers. Last, it checks that you are not returning too many k-mers.

Input:

```
ATA
3 1
```

Output:

```
AAA AAT ACA AGA ATA ATC ATG ATT CAT CTA GAT GTA TAA TAC TAG TAT TCT TGT TTA
TTT
```

Case 6

Description: This dataset checks that your code is looking at *both Text* and its reverse complement (i.e. not just looking at *Text*, and not just looking at the reverse complement of *Text*, but looking at both).

Input:

```
AAT
3 0
```

Output:

```
AAT ATT
```

Case 7

Description: This dataset checks that your code correctly delimiting your output (i.e. using spaces) and verifies that your k-mers are actually of length k.

Input:

```
TAGCG
2 1
```

Output:

```
CA CC GG TG
```

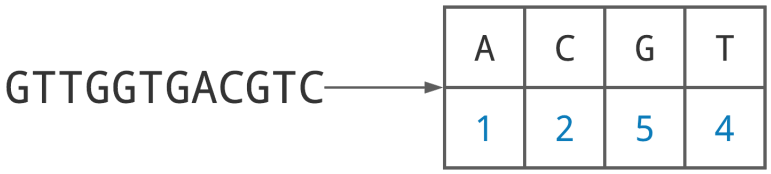
Case 8

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

1K Generate the Frequency Array of a String

Frequency Array Problem
Generate the frequency array of a DNA string.

Input: A DNA string *Text* and an integer *k*.
Output: The frequency array of *k*-mers in *Text*.



Formatting

Input: A DNA string *Text* and an integer *k*.
Output: A space-separated list of integers representing the frequency array of *k*-mers in *Text*).

Constraints

- The length of *Text* will be between 1 and 10^3.
- The integer *k* will be between 1 and 10^1.
- *Text* will be a DNA string.

Test Cases

Case 1

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

Input:

```
ACGCGGCTCTGAAA
2
```

Output:

```
2 1 0 0 0 0 2 2 1 2 1 0 0 1 1 0
```

Case 2

Description: This dataset checks if you have an off-by-one error at the end of *Text* (i.e. you are not counting the last *k*-mer in *Text*). There are three instances of AA (**AA**AAC, A**AA**AC, and AA**AA**C), but there is one instance of AC at the end (AAA**AC**).

Input:

```
AAAAC
2
```

Output:

```
3 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Case 3

Description: This dataset checks if you have an off-by-one error at the beginning of *Text* (i.e. you are not counting the first *k*-mer in *Text*). There are two instances of AA (TT**AA**A and TTA**AA**), but there is one instance of TA (T**TA**AA) and one instance of TT (**TT**AAA).

Input:

```
TTAAA
2
```

Output:

```
2 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1
```

Case 4

Description: This dataset checks if your code actually increments each count, or if your code instead just sets the count equal to one each time. In other words, this dataset checks if your code is doing something like array[k-mer]$= 1$ instead of array[k-mer]$+ = 1$.

Input:

```
AAA
2
```

Output:

```
2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

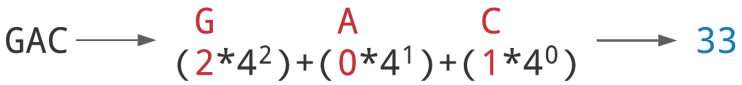
Case 5

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

1L Implement PatternToNumber

PatternToNumber Problem
Convert a DNA string to a number.

Input: A DNA string *Pattern*.
Output: PATTERNTONUMBER(*Pattern*).



Formatting

Input: A DNA string *Pattern*.
Output: An integer representing the output of PATTERNTONUMBER(*Pattern*).

Constraints

- The length of *Pattern* will be between 1 and 10^2.
- *Pattern* will be a DNA string.

Test Cases

Case 1

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

Input:
```
AGT
```

Output:
```
11
```

Case 2

Description: *Pattern* is made up of only one character.

Input:
```
CCC
```

Output:
```
21
```

Case 3

Description: *Pattern* is long, but is 'A'-dense.

Input:
```
AAAAAAAAAAG
```

Output:
```
2
```

Case 4

Description: *Pattern* has a length of 1.

Input:
```
T
```

Output:
```
3
```

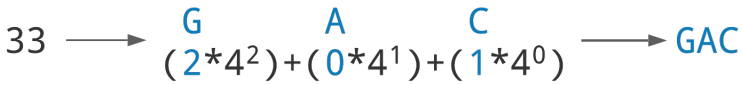
Case 5

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

1M Implement NumberToPattern

NumberToPattern Problem
Convert a number to its corresponding DNA string.

Input: Integers *index* and *k*.
Output: NUMBERTOPATTERN(*index*, *k*).



Formatting

Input: Space-separated integers *index* and *k*.
Output: A string representing the output of *NumberToPattern*(*index*, *k*).

Constraints

- The integer *index* will be between 1 and 10^4.
- The integer *k* will be between 1 and 10^1.

Test Cases

Case 1

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

Input:
```
45 4
```

Output:
```
AGTC
```

Case 2

Description: *k* is small.

Input:
```
1 8
```

Output:
```
G
```

Case 3

Description: *k* codes for an empty string

Input:
```
0 0
```

Output:

Case 4

Description: *k* is large.

Input: Space-separated integers *k* and *d* followed by a space-separated list of paired *k*-mer strings *PairedReads* where individual *k*-mers within the pair are separated by a "|" character.
Input:
```
60 4
```

Output:
```
ATTA
```

Case 5

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

1N Generate the d-Neighborhood of a String

***d*-Neighborhood Problem**
Find all the neighbors of a pattern.

Input: A DNA string *Pattern* and an integer d.
Output: The collection of strings NEIGHBORS(*Pattern*, d).

CGA AAA AGC

AGA GGA ACA AGG

TGA ATA AGT

Formatting

Input: A DNA string *Pattern* and an integer d.
Output: A space-separated list of strings containing all *Neighbors*(*Pattern*, d).

Constraints

- The length of *Pattern* will be between 1 and 10^1.
- The integer d will be between 1 and 10^1.
- *Pattern* will be a DNA string.

Test Cases

Case 1

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

Input:

```
ACG
1
```

Output:

```
AAG ACA ACC ACG ACT AGG ATG CCG GCG TCG
```

Case 2

Description: $d = 0$.

Input:

```
AGA
0
```

Output:

```
AGA
```

Case 3

Description: *Pattern* is made up of one character.

Input:

```
AAA
1
```

Output:

```
AAA AAC AAG AAT ACA AGA ATA CAA GAA TAA
```

Case 4

Description: *Pattern* has a length of 1.

Input:

```
A
1
```

Output:

```
A C G T
```

Case 5

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

1O Implement ApproximatePatternCount

Approximate Pattern Count Problem
Count all approximate occurrences of a pattern in a string.

Input: Strings *Pattern* and *Text* as well as an integer d.
Output: $\text{COUNT}_d(Text, Pattern)$.

```
CGACTAGTTT
CGACGA
1    2
```

Formatting

Input: A DNA string *Pattern* followed by a DNA string *Text*, followed by an integer d.
Output: A single integer $\text{COUNT}_d(Text, Pattern)$.

Constraints

- The length of *Pattern* will be between 1 and 10^1.
- The length of *Text* will be between 1 and 10^3
- The integer d will be between 1 and 10^1.
- Both *Pattern Text* and will be DNA strings.

Test Cases

Case 1

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

Input:

```
GAGG
TTTAGAGCCTTCAGAGG
2
```

Output:

```
4
```

Case 2

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

2 Which DNA Patterns Play the Role of Molecular Clocks?

Randomized Algorithms

2A Implement MotifEnumeration

Implanted Motif Problem
Find all (k, d)-motifs in a collection of strings.

Input: A collection of strings *Dna*, and integers *k* and *d*.
Output: All (*k*, *d*)-motifs in *Dna*.

AAAGGT
CCGCTA
TGGACT
AGTAAC

Formatting

Input: Space-separated integers *k* and *d*, followed by a space-separated collection of strings *Dna*.
Output: A space-separated list of strings representing all (*k*, *d*)-motifs in *Dna*.

Constraints

- The integer *k* will be between 1 and 10^1.
- The integer *d* will be between 1 and 10^1.
- The number of strings in *Dna* will be between 1 and 10^2.
- Each string in *Dna* will have a length between 1 and 10^2.
- All strings in *Dna* will be DNA strings.

Test Cases

Case 1

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

Input:

```
3 1
ATTTGGC TGCCTTA CGGTATC GAAAATT
```

Output:

```
ATA ATT GTT TTT
```

Case 2

Description: This dataset checks for off-by-one errors, both at the beginning and at the end. The 3-mers `ACG` and `CGT` both appear perfectly in all 3 strings in *Dna*. Thus, if your output doesn't contain `ACG`, you are most likely not counting the first k-mer of every string. Similarly, if your output doesn't contain `CGT`, you are most likely not counting the last k-mer of every string.

Input:

```
3 0
ACGT ACGT ACGT
```

Output:

```
ACG CGT
```

Case 3

Description: This dataset checks if your code work correctly when $d > 0$. If your code only counts motifs with $d = 0$ (and not $d > 0$), your code will only find a single motif (`AAA`, which is the only 3-mer that occurs perfectly in all of the strings of *Dna*). A correct solution would, in addition to `AAA`, find all 3-mers that differ from `AAA` by exactly 1 base.

Input:

```
3 1
AAAAA AAAAA AAAAA
```

Output:

```
AAA AAC AAG AAT ACA AGA ATA CAA GAA TAA
```

Case 4

Description: This dataset checks if your code counts motifs where the number of mismatches is equal to d in addition to motifs where the number of mismatches is less than d. For example, in this dataset, a correct solution would find *all* 3-mers (because we are allowing for 3 mismatches). However, an incorrect solution that counts mismatches less than d but not mismatches equal to d would only find the k-mers that differ from `AAA` by 1 or 2 bases, not the ones that differ from `AAA` by 3 bases.

Input:

```
3 3
AAAAA AAAAA AAAAA
```

Output:

```
AAA AAC AAG AAT ACA ACC ACG ACT AGA AGC AGG AGT ATA ATC ATG ATT CAA CAC CAG
CAT CCA CCC CCG CCT CGA CGC CGG CGT CTA CTC CTG CTT GAA GAC GAG GAT GCA GCC
GCG GCT GGA GGC GGG GGT GTA GTC GTG GTT TAA TAC TAG TAT TCA TCC TCG TCT TGA
TGC TGG TGT TTA TTC TTG TTT
```

Case 5

Description: This test dataset checks if your code is checking the last sequence in *Dna*. If your code only checks the first two sequences in the dataset, the 3-mer `AAA` exists perfectly in both and will thus be output. If your code checks the last sequence of *Dna* (`AACAA`), however, it will find that `AAA` does not appear. Thus, `AAA` is not a motif in *Dna*, and no sequences should be output.

Input:

```
3 0
AAAAA AAAAA AACAA
```

Output:

Case 6

Description: This test dataset checks if your code is checking the first sequence in *Dna*. If your code only checks the last two sequences in the dataset, the 3-mer `AAA` exists perfectly in both and will thus be output. If your code checks the first sequence of *Dna* (`AACAA`), however, it will find that `AAA` does not appear. Thus, `AAA` is not a motif in *Dna*, and no sequences should be output.

Input:

```
3 0
AACAA AAAAA AAAAA
```

Output:

Case 7

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

2B Find a Median String

Median String Problem
Find a median string.

Input: A collection of strings *Dna* and an integer *k*.
Output: A *k*-mer *Pattern* that minimizes D(*Pattern*, *Dna*) among all possible choices of *k*-mers.

```
CTTAAC
GATATC   AAA
ACGGCG
CTAAAG
```

Formatting

Input: An integer *k*, followed by a newline-separated collection of strings *Dna*.
Output: A string representing a *k*-mer *Pattern* that minimizes *d*(*Pattern*, *Dna*) over all *k*-mers *Pattern* (If multiple answers exist, you may return any one).

Constraints

- The integer *k* will be between 1 and 10^1.
- The number of strings in *Dna* will be between 1 and 10^2.
- The length of each string in *Dna* will be between 1 and 10^2.
- Each string in *Dna* will be a DNA string.

Test Cases

Case 1

Description: The sample dataset is not actually run on your code. Notice that there are technically two solutions to the problem (`ACG` and `GAC` are equally optimal), but the problem specifically states to only return a single output (you can arbitrarily pick any optimal solution).

Input:

```
3
AAATTGACGCAT GACGACCACGTT CGTCAGCGCCTG GCTGAGCACCGG AGTACGGGACAG
```

Output:

```
ACG
```

Case 2

Description: This dataset checks that your output is the correct length. Notice that there are technically two solutions to the problem (`ACG` and `CGT` are equally optimal), but the problem specifically states to only return a single output (you can arbitrarily pick any optimal solution). Also, since $k = 3$ in this dataset, we check that your output is exactly of length k (should not be any longer or shorter than this).

Input:

```
3
ACGT ACGT ACGT
```

Output:

```
ACG
```

Case 3

Description: This dataset checks if your code considers k-mers that do not occur in *Dna*. Notice that the best 3-mer is `AAA`, which does not actually occur in any of the sequences in *Dna*. It is perfectly fine that our optimal median string does not actually occur in any of the sequences in *Dna* (similar to the Frequent Words With Mismatches Problem from chapter one).

Input:

```
3
ATA ACA AGA AAT AAC
```

Output:

```
AAA
```

Case 4

Description: This dataset checks that your output only contains a single *k*-mer. Notice that there are technically two solutions to the problem (`AAG` and `AAT` are equally optimal), but the problem specifically states to only return a single output (you can arbitrarily pick any optimal solution).

Input:

```
3
AAG AAT
```

Output:

```
AAG
```

Case 5

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

2C Find a Profile-most Probable *k*-mer in a String

Profile-most Probable *k*-mer Problem
Find a profile-most probable k-mer in a string.

Input: A string *Text*, an integer k, and a $4 \times k$ matrix *Profile*.
Output: A *Profile*-most probable k-mer in *Text*.

	C	A	G	A	T	G	T	C	T	G
A	6/10	1/10	2/10	0/10	1/10					
C	2/10	1/10	4/10	0/10	7/10					
G	0/10	0/10	2/10	2/10	0/10					
T	2/10	8/10	2/10	8/10	2/10					
		A	T	C	T	C				

Formatting

Input: A string *Text*, an integer k, and a $4 \times k$ matrix *Profile* of floats.
Output: A string representing a *Profile*-most probable k-mer in *Text* (If multiple answers exist, you may return any one).

Constraints

- The length of *Text* will be between 1 and 10^3.
- The integer k will be between 1 and 10^1.
- *Text* will be a DNA string.

Test Cases

Case 1

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

Input:

```
ACCTGTTTATTGCCTAAGTTCCGAACAAACCCAATATAGCCCGAGGGCCT
5
0.2 0.2 0.3 0.2 0.3
0.4 0.3 0.1 0.5 0.1
0.3 0.3 0.5 0.2 0.4
0.1 0.2 0.1 0.1 0.2
```

Output:

```
CCGAG
```

Case 2

Description: This dataset checks for off-by-one errors at the beginning of *Text*. Notice that the optimal solution (`AGCAGCTT`) occurs at the very beginning of *Text*, so if your code does not check this *k*-mer, then your code will output a different (incorrect) *k*-mer as the solution.

Input:

```
AGCAGCTTTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTCTGATCTGAACTGGT...
...TACCTGCCGTGAGTAAAT
8
0.7 0.2 0.1 0.5 0.4 0.3 0.2 0.1
0.2 0.2 0.5 0.4 0.2 0.3 0.1 0.6
0.1 0.3 0.2 0.1 0.2 0.1 0.4 0.2
0.0 0.3 0.2 0.0 0.2 0.3 0.3 0.1
```

Output:

```
AGCAGCTT
```

Case 3

Description: This dataset checks for off-by-one errors at the end of *Text*. Notice that the optimal solution (AAGCAGAGTTTA) occurs at the very end of *Text*, so if your code does not check this *k*-mer, then your code will output a different (incorrect) *k*-mer as the solution.

Input:

```
TTACCATGGGACCGCTGACTGATTTCTGGCGTCAGCGTGATGCTGGTGTGGATGACATTCCGGTGCGCTT...
...TGTAAGCAGAGTTTA
12
0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.1 0.2 0.3 0.4 0.5
0.3 0.2 0.1 0.1 0.2 0.1 0.1 0.4 0.3 0.2 0.2 0.1
0.2 0.1 0.4 0.3 0.1 0.1 0.1 0.3 0.1 0.1 0.2 0.1
0.3 0.4 0.1 0.1 0.1 0.1 0.0 0.2 0.4 0.4 0.2 0.3
```

Output:

```
AAGCAGAGTTTA
```

Case 4

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

2D Implement GreedyMotifSearch

Greedy Motif Search Problem
Implement GreedyMotifSearch.

Input: A collection of strings *Dna*, and integers k and t.
Output: A collection of strings resulting from running GREEDYMOTIFSEARCH(*Dna*, k, t).

```
tACCTtaa
ATGTctgt
cgGCGTta
tcagAGGT
ctaACGAg
```

Formatting

Input: Space-separated integers k and t, followed by a newline-separated collection of strings *Dna*.
Output: A space-separated list of strings resulting from running GREEDYMOTIFSEARCH(*Dna*, k, t) (If at any step you find more than one *Profile*-most probable k-mer in a given string, use the one occurring first).

Constraints

- The integer k will be between 1 and 10^2.
- The integer t will be between 1 and 10^2.
- The number of strings in *Dna* will be between 1 and 10^2.
- The length of each string in *Dna* will be between 1 and 10^2.
- Each string in *Dna* will be a DNA string.

Test Cases

Case 1

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

Input:

```
3 5
GGCGTTCAGGCA AAGAATCAGTCA CAAGGAGTTCGC CACGTCAATCAC CAATAATATTCG
```

Output:

```
CAG CAG CAA CAA CAA
```

Case 2

Description: This dataset checks that your code always picks the first-occurring Profile-most Probable *k*-mer in a given sequence of *Dna*. In the first sequence (`GCCCAA`), `GCC` and `CCA` are both Profile-most Probable *k*-mers. However, you must return `GCC` since it occurs earlier than `CCA`. Thus, if the first sequence of your output is `CCA`, this test case fails your code.

Input:

```
3 4
GCCCAA GGCCTG AACCTA TTCCTT
```

Output:

```
GCC GCC AAC TTC
```

Case 3

Description: This dataset checks if your code has an off-by-one error at the beginning of each sequence of *Dna*. Notice that the first four motifs of the solution occur at the beginning of their respective sequences in *Dna*, so if your code did not check the first *k*-mer in each sequence of *Dna*, it would not find these sequences.

Input:

```
5 8
GAGGCGCACATCATTATCGATAACGATTCGCCGCATTGCC
TCATCGAATCCGATAACTGACACCTGCTCTGGCACCGCTC
TCGGCGGTATAGCCAGAAAGCGTAGTGCCAATAATTTCCT
GAGTCGTGGTGAAGTGTGGGTTATGGGGAAAGGCAGACTG
GACGGCAACTACGGTTACAACGCAGCAACCGAAGAATATT
TCTGTTGTTGCTAACACCGTTAAAGGCGGCGACGGCAACT
AAGCGGCCAACGTAGGCGCGGCTTGGCATCTCGGTGTGTG
AATTGAAAGGCGCATCTTACTCTTTTCGCTTTCAAAAAAA
```

Output:

```
GAGGC TCATC TCGGC GAGTC GCAGC GCGGC GCGGC GCATC
```

Case 4

Description: This dataset checks if your code has an off-by-one error at the end of each sequence of *Dna*. Notice that the first two motifs of the solution occur at the end of their respective sequences in *Dna*, so if your code did not check the end *k*-mer in each sequence of *Dna*, it would not find these sequences.

Input:

```
6 5
GCAGGTTAATACCGCGGATCAGCTGAGAAACCGGAATGTGCGT
CCTGCATGCCCGGTTTGAGGAACATCAGCGAAGAACTGTGCGT
GCGCCAGTAACCCGTGCCAGTCAGGTTAATGGCAGTAACATTT
AACCCGTGCCAGTCAGGTTAATGGCAGTAACATTTATGCCTTC
ATGCCTTCCGCGCCAATTGTTCGTATCGTCGCCACTTCGAGTG
```

Output:

```
GTGCGT GTGCGT GCGCCA GTGCCA GCGCCA
```

Case 5

Description: This test dataset checks if your code is correctly breaking ties when calling Profile-most Probable *k*-mer. Specifically, it makes sure that, when you call Profile-most Probable *k*-mer, in the event of a tie, you choose the first-occurring *k*-mer.

Input:

```
5 8
GACCTACGGTTACAACGCAGCAACCGAAGAATATTGGCAA
TCATTATCGATAACGATTCGCCGGAGGCCATTGCCGCACA
GGAGTCTGGTGAAGTGTGGGTTATGGGGCAGACTGGGAAA
GAATCCGATAACTGACACCTGCTCTGGCACCGCTCTCATC
AAGCGCGTAGGCGCGGCTTGGCATCTCGGTGTGTGGCCAA
AATTGAAAGGCGCATCTTACTCTTTTCGCTTAAAATCAAA
GGTATAGCCAGAAAGCGTAGTTAATTTCGGCTCCTGCCAA
TCTGTTGTTGCTAACACCGTTAAAGGCGGCGACGGCAACT
```

Output:

```
GCAGC TCATT GGAGT TCATC GCATC GCATC GGTAT GCAAC
```

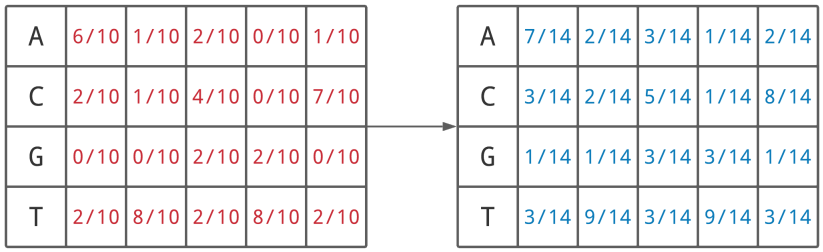
Case 6

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

2E Implement GreedyMotifSearch with Pseudocounts

Greedy Motif Search with Pseudocounts Problem
Implement GreedyMotifSearch with pseudocounts.

Input: A collection of strings *Dna*, and integers k and t.
Output: A collection of strings resulting from running GREEDYMOTIFSEARCH(*Dna*, k, t) with pseudocounts.



A	6/10	1/10	2/10	0/10	1/10
C	2/10	1/10	4/10	0/10	7/10
G	0/10	0/10	2/10	2/10	0/10
T	2/10	8/10	2/10	8/10	2/10

A	7/14	2/14	3/14	1/14	2/14
C	3/14	2/14	5/14	1/14	8/14
G	1/14	1/14	3/14	3/14	1/14
T	3/14	9/14	3/14	9/14	3/14

Formatting

Input: Space-separated integers k and t, followed by a newline-separated collection of strings *Dna*.
Output: A space-separated list of strings containing a collection of strings resulting from running GREEDYMOTIFSEARCH(*Dna*, k, t) with pseudocounts (If at any step you find more than one *Profile*-most probable k-mer in a given string, use the one occurring first).

Constraints

- The integer k will be between 1 and 10^2.
- The integer t will be between 1 and 10^2.
- The number of strings in *Dna* will be between 1 and 10^2.
- The length of each string in *Dna* will be between 1 and 10^2.
- Each string in *Dna* will be a DNA string.

Test Cases

Case 1

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

Input:

```
3 5
GGCGTTCAGGCA AAGAATCAGTCA CAAGGAGTTCGC CACGTCAATCAC CAATAATATTCG
```

Output:

```
TTC ATC TTC ATC TTC
```

Case 2

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

2F Implement RandomizedMotifSearch

Randomized Motif Search Problem
Implement RandomizedMotifSearch.

Input: A collection of strings *Dna*, and integers k and t.
Output: A collection of strings resulting from running RANDOMIZEDMOTIFSEARCH(*Dna*, k, t) 1000 times. Remember to use pseudocounts!

Formatting

Input: Space-separated integers k and t, followed by a newline-separated collection of strings *Dna*.
Output: A space-separated list of strings containing a collection of strings resulting from running RANDOMIZEDMOTIFSEARCH(*Dna*, k, t) 1000 times. Remember to use pseudocounts!

Constraints

- The integer k will be between 1 and 10^2.
- The integer t will be between 1 and 10^2.
- The number of strings in *Dna* will be between 1 and 10^2.
- The length of each string in *Dna* will be between 1 and 10^2.
- Each string in *Dna* will be a DNA string.

Test Cases

Case 1

Description: A small and hand-solvable dataset taken from the example problem on Stepik.

Input:

```
8 5
CGCCCCTCTCGGGGGTGTTCAGTAAACGGCCA GGGCGAGGTATGTGTAAGTGCCAAGGTGCCAG
TAGTACCGAGACCGAAAGAAGTATACAGGCGT TAGATCAAGTTTCAGGTGCACGTCGGTGAACC
AATCCACCAGCTCCACGTGCAATGTTGGCCTA
```

Output:

```
TCTCGGGG CCAAGGTG TACAGGCG TTCAGGTG TCCACGTG
```

Case 2

Description: This dataset checks if your code has an off-by-one error at the beginning of each sequence of *Dna*. Notice that the some of the motifs of the solution occur at the beginning of their respective sequences in *Dna*, so if your code did not check the first *k*-mer in each sequence of *Dna*, it would not find these sequences.

Input:

```
6 8
AATTGGCACATCATTATCGATAACGATTCGCCGCATTGCC
GGTTAACATCGAATAACTGACACCTGCTCTGGCACCGCTC
AATTGGCGGCGGTATAGCCAGATAGTGCCAATAATTTCCT
GGTTAATGGTGAAGTGTGGGTTATGGGGAAAGGCAGACTG
AATTGGACGGCAACTACGGTTACAACGCAGCAAGAATATT
GGTTAACTGTTGTTGCTAACACCGTTAAGCGACGGCAACT
AATTGGCCAACGTAGGCGCGGCTTGGCATCTCGGTGTGTG
GGTTAAAAGGCGCATCTTACTCTTTTCGCTTTCAAAAAAA
```

Output:

```
CGATAA GGTTAA GGTATA GGTTAA GGTTAC GGTTAA GGCCAA GGTTAA
```

Case 3

Description: This dataset checks if your code has an off-by-one error at the end of each sequence of *Dna*. Notice that the some of the motifs of the solution occur at the end of their respective sequences in *Dna*, so if your code did not check the last *k*-mer in each sequence of *Dna*, it would not find these sequences.

Input:

```
6 8
GCACATCATTAAACGATTCGCCGCATTGCCTCGATTAACC
TCATAACTGACACCTGCTCTGGCACCGCTCATCCAAGGCC
AAGCGGGTATAGCCAGATAGTGCCAATAATTTCCTTAACC
AGTCGGTGGTGAAGTGTGGGTTATGGGGAAAGGCAAGGCC
AACCGGACGGCAACTACGGTTACAACGCAGCAAGTTAACC
AGGCGTCTGTTGTTGCTAACACCGTTAAGCGACGAAGGCC
AAGCTTCCAACATCGTCTTGGCATCTCGGTGTGTTTAACC
AATTGAACATCTTACTCTTTTCGCTTTCAAAAAAAAGGCC
```

Output:

```
TTAACC ATAACT TTAACC TGAAGT TTAACC TTAAGC TTAACC TGAACA
```

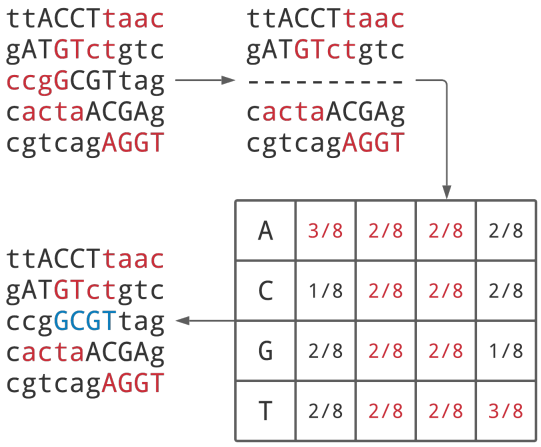
Case 4

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

2G Implement GibbsSampler

Gibbs Sampler Problem
Implement GibbsSampler.

Input: A collection of DNA strings *Dna*, and integers k, t, and N.
Output: The strings resulting from running GIBBSSAMPLER(*Dna*, k, t, N) with 20 random starts. Remember to use pseudocounts!



Formatting

Input: Space-separated integers k, t, and N, followed by a newline-separated collection of DNA strings *Dna*.
Output: A space-separated list of strings containing the strings resulting from running GIBBSSAMPLER(*Dna*, k, t, N) with 20 random starts. Remember to use pseudocounts!

Constraints

- The integer k will be between 1 and 10^2.
- The integer t will be between 1 and 10^2.
- The integer N will be between 1 and 10^4.
- The number of strings in *Dna* will be between 1 and 10^2.
- The length of each string in *Dna* will be between 1 and 10^3.
- Each string in *Dna* will be a DNA string.

Test Cases

Case 1

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

Input:

```
8 5 100
CGCCCCTCTCGGGGGTGTTCAGTAAACGGCCA GGGCGAGGTATGTGTAAGTGCCAAGGTGCCAG
TAGTACCGAGACCGAAAGAAGTATACAGGCGT TAGATCAAGTTTCAGGTGCACGTCGGTGAACC
AATCCACCAGCTCCACGTGCAATGTTGGCCTA
```

Output:

```
TCTCGGGG CCAAGGTG TACAGGCG TTCAGGTG TCCACGTG
```

Case 2

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

2H Implement DistanceBetweenPatternAndStrings

Distance Between Pattern and Strings Problem
Compute DistanceBetweenPatternAndStrings.

Input: A DNA string *Pattern* and a collection of DNA strings *Dna*.
Output: Distance D(*Pattern*, *Dna*) between *Pattern* and *Dna*.

$$d(\text{AAA}, \begin{matrix}\text{CTTAAC}\\ \text{GATATC}\\ \text{ACGGCG}\\ \text{CTAAAG}\end{matrix}) = 4$$

Formatting

Input: A DNA string *Pattern*, followed by a space-separated collection of DNA strings *Dna*.
Output: An integer representing the output of DISTANCEBETWEENPATTERNANDSTRINGS(*Pattern*, *Dna*).

Constraints

- The length of *Pattern* will be between 1 and 10^1.
- The number of strings in *Dna* will be between 1 and 10^2.
- The length of each string in *Dna* will be between 1 and 10^2.
- *Pattern* and each string in *Dna* will be DNA strings.

Test Cases

Case 1

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

Input:

```
AAA
TTACCTTAAC GATATCTGTC ACGGCGTTCG CCCTAAAGAG CGTCAGAGGT
```

Output:

```
5
```

Case 2

Description: This dataset checks multiple potential mistakes. First, it checks that you are actually using all three sequences of *Dna* (and not just a single sequence). The Hamming Distance between *Pattern* and each individual sequence in *Dna* is 1, so if your code returns a total score of 1, we fail it for this reason. Next, it checks if you are only using the first k-mer in each sequence of Dna. For example, if you do this, you would output $d(\texttt{TAA},\texttt{TTT})+d(\texttt{TAA},\texttt{CCT})+d(\texttt{TAA},\texttt{GGT})$ which is 8, instead of the correct answer of 3. Finally, it checks if you are only using the last k-mer in each sequence of *Dna*. For example, if you do this, you would output $d(\texttt{TAA},\texttt{TTT})+d(\texttt{TAA},\texttt{CAC})+d(\texttt{TAA},\texttt{GAG})$ which is 6, instead of the correct answer of 3.

Input:

```
TAA
TTTATTT CCTACAC GGTAGAG
```

Output:

```
3
```

Case 3

Description: This dataset checks if your code is using maximum or sum instead of minimum. First, it checks if your code is using maximum instead of minimum. In this case, the output would be $d(\texttt{AAA},\texttt{ACT})+d(\texttt{AAA},\texttt{AAC})+d(\texttt{AAA},\texttt{AAG})$, which is 4, instead of the correct answer of 0. Next, it checks if your code is using sum instead of minimum. In this case, the output would be $d(\texttt{AAA},\texttt{AAA})+d(\texttt{AAA},\texttt{AAC})+d(\texttt{AAA},\texttt{ACT})+d(\texttt{AAA},\texttt{AAA})+d(\texttt{AAA},\texttt{AAC})+d(\texttt{AAA},\texttt{AAA})+d(\texttt{AAA},\texttt{AAG})$, which is 5, instead of the correct answer of 0.

Input:

```
AAA
AAACT AAAC AAAG
```

Output:

```
0
```

Case 4

Description: This dataset checks if your code has an off-by-one error at the end of each sequence of *Dna*. Notice that each sequence has a perfect match of `AAA` at the very end, so if your code returns a nonzero answer to this test dataset, it must have missed the last *k*-mer of each.

Input:

```
AAA
TTTTAAA CCCCAAA GGGGAAA
```

Output:

```
0
```

Case 5

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

3 How Do We Assemble Genomes?

Graph Algorithms

3A Generate the k-mer Composition of a String

String Composition Problem
Generate the k-mer composition of a string.

Input: An integer k and a string *Text*.
Output: The collection of k-mers COMPOSITION$_k$(*Text*).

```
CATGGGTG
CA
 AT
  TG
   GG
    GG
     GT
      TG
```

Formatting

Input: A integer k followed by a string *Text*.
Output: A space-separated list of k-mer strings representing COMPOSITION$_k$(*Text*) (the k-mers can be provided in any order).

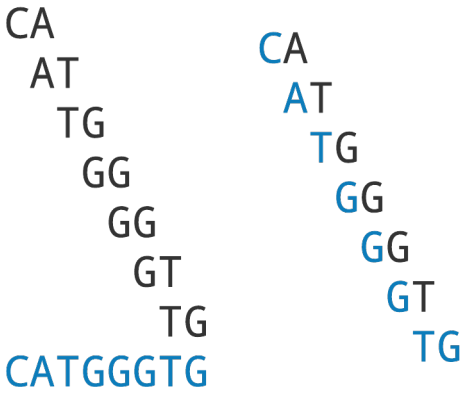
Constraints

- The value of k will be between 1 and 10^3.
- The length *Text* will be between 1 and 10^4.
- No pattern is a prefix of another pattern.

3B Reconstruct a String from its Genome Path

String Spelled by a Genome Path Problem
Find the string spelled by a genome path.

Input: A sequence of k-mers $Pattern_1, \ldots, Pattern_n$ such that the last $k-1$ symbols of $Pattern_i$ are equal to the first $k-1$ symbols of $Pattern_i + 1$ for i from 1 to $n-1$.
Output: A string *Text* of length $k+n-1$ where the i-th k-mer in *Text* is equal to $Pattern_i$ for all i from 1 to n.



Formatting

Input: A space-separated list of strings *Patterns*.
Output: A string *Text*.

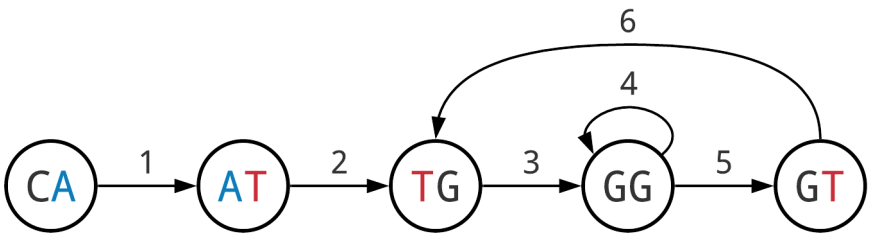
Constraints

- The number of patterns in the string-set *Pattern* will be between 1 and 10^4.
- The length of any one pattern in *Pattern* will be between 1 and 10^2.

3C Construct the Overlap Graph of a Collection of *k*-mers

Overlap Graph Problem
Construct the overlap graph of a collection of k-mers.

Input: A collection *Patterns* of *k*-mers.
Output: The overlap graph of *Patterns*.



Formatting

Input: A space-separated list of strings *Patterns*.
Output: An adjacency list representing the overlap graph of *Patterns*.

Constraints

- The number of patterns in the string-set *Patterns* will be between 1 and 10^3.
- The length of any one pattern in *Patterns* will be between 1 and 10^2.

3D Construct the De Bruijn Graph of a String

De Bruijn Graph from a String Problem
Construct the de Bruijn graph of a string.

Input: An integer k and a string *Text*.
Output: The graph DEBRUIJN$_k$(*Text*).

c3/logos/3D.png

Formatting

Input: A integer k followed by a string *Text*.
Output: DEBRUIJN$_k$(*Text*), in the form of an adjacency list.

Constraints

- The value of k will be between 1 and 10^2.
- The length of *Text* will be between 1 and 10^4.

3E Construct the De Bruijn Graph of a Collection of k-mers

De Bruijn Graph from k-mers Problem
Construct the de Bruijn graph from a collection of k-mers.

Input: A collection of k-mers *Patterns*.
Output: The graph DEBRUIJN(*Patterns*).

c3/logos/3E.png

Formatting

Input: A space-separated list of k-mer strings *Patterns*.
Output: An adjacency list representing DEBRUIJN(*Patterns*).

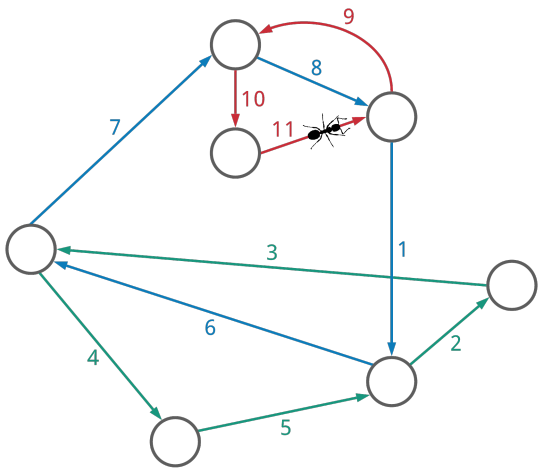
Constraints

- The number of patterns in the string-set *Patterns* will be between 1 and 10^4.
- The length of any one pattern in *Patterns* will be between 1 and 10^2.
- All strings in *Patterns* will be DNA strings.

3F Find an Eulerian Cycle in a Graph

Eulerian Cycle Problem
Find an Eulerian cycle in a graph.

Input: An Eulerian directed graph.
Output: An Eulerian cycle in this graph.



Formatting

Input: An adjacency list representing an Eulerian directed graph.
Output: A space-separated list of integers representing an Eulerian cycle in the directed graph.

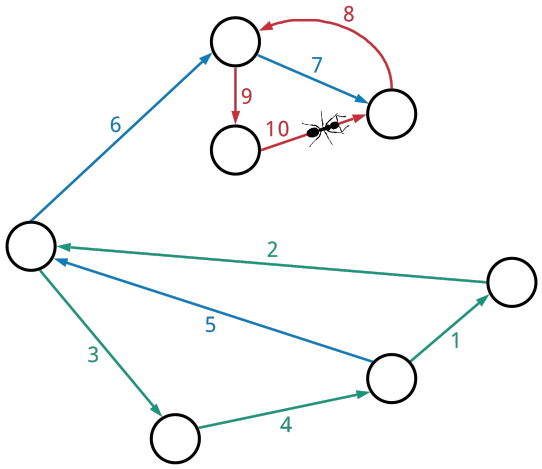
Constraints

- The number of nodes in the graph will be between 1 and 10^4.
- The number of edges in the graph will be between 1 and 10^4.
- All nodes in the graph will be labeled with integers.

3G Find an Eulerian Path in a Graph

Eulerian Path Problem
Find an Eulerian path in a graph.

Input: A directed graph containing an Eulerian path.
Output: An Eulerian path in this graph.



Formatting

Input: An adjacency list representing a directed graph containing an Eulerian path.
Output: A space-separated list of integers representing an Eulerian path in the graph.

Constraints

- The number of nodes in the graph will be between 1 and 10^4.
- The number of edges in the graph will be between 1 and 10^4.
- All nodes in the graph will be labeled with integers.

3H Reconstruct a String from its k-mer Composition

String Reconstruction Problem
Reconstruct a string from its k-mer composition.

Input: An integer k and a collection of k-mers *Patterns*.
Output: A string *Text* with k-mer composition equal to *Patterns*.

c3/logos/3H.png

Formatting

Input: An integer k followed by a space-separated list of k-mer strings *Patterns*.
Output: A string *Text* with k-mer composition equal to *Patterns* (if multiple answers exist, you may return any one).

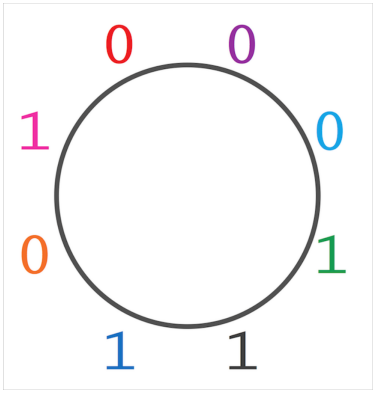
Constraints

- The value of k will be between 1 and 10^2.
- The number of strings in *Patterns* will be between 1 and 10^4.
- The length of any one pattern in *Pattern* will be between 1 and 10^2.

3I Find a k-Universal Circular String

k-Universal Circular String Problem
Find a k-universal circular binary string.

Input: An integer k.
Output: A k-universal circular binary string.



Formatting

Input: An integer k.
Output: A binary string representing a k-universal circular string.

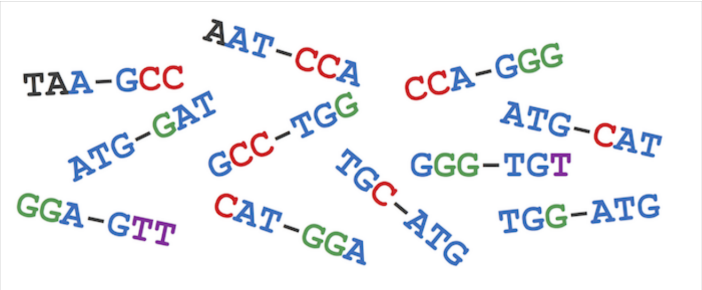
Constraints

- The value of k will be between 1 and 10^1.

3J Reconstruct a String from its Paired Composition

String Reconstruction from Read-Pairs Problem
Reconstruct a string from its paired composition.

Input: Integers k and d and a collection of paired k-mers *PairedReads*.
Output: A string *Text* with (k,d)-mer composition equal to *PairedReads*.



Formatting

Input: Space-separated integers k and d followed by a space-separated list of paired k-mer strings *PairedReads* where individual k-mers within the pair are separated by a "|" character.
Output: A string *Text* with (k,d)-mer composition equal to *PairedReads* (if multiple answers exist, you may return any one).

Constraints

- The value of k will be between 1 and 10^2.
- The value of d will be between 1 and 10^3.
- The number of strings in *PairedReads* will be between 1 and 10^4.
- The length of any one pair of paired k-mers in *PairedReads* will be between 1 and 10^2.
- All k-mer strings in *PairedReads* will be DNA strings.

3K Generate Contigs from a Collection of Reads

Contig Generation Problem
Generate the contigs from a collection of reads (with imperfect coverage).

Input: A collection of k-mers *Patterns*.
Output: All contigs in the graph DEBRUIJN(*Patterns*).

c3/logos/3K.png

Formatting

Input: A space-separated list of k-mer strings *Patterns*.
Output: A space-separated list of strings representing all contigs in the graph DEBRUIJN(*Patterns*) (you may return the strings in any order).

Constraints

- The number of strings in *Patterns* will be between 1 and 10^4.
- The length of any given string in *Patterns* will be between 1 and 10^2.

3L Construct a String Spelled by a Gapped Genome Path

Gapped Genome Path String Problem
Reconstruct a string from a sequence of (k,d)-mers corresponding to a path in a paired de Bruijn graph.

Input: A collection of (k, d)-mers $(a_1 \mid b_1)$, ... , $(a_n \mid b_n)$ such that SUFFIX$(a_i \mid b_i)$ = PREFIX$(a_{i+1} \mid b_{i+1})$ for all i from 1 to $n - 1$, *PairedReads*.
Output: A string *Text* where the i-th k-mer in *Text* is equal to SUFFIX$(a_i \mid b_i)$ for all i from 1 to n, if such a string exists.

c3/logos/3L.png

Formatting

Input: Space-separated integers k and d followed by a space-separated list of paired k-mer strings *PairedReads* where individual k-mers within the pair are separated by a "|" character.
Output: A string *Text* where the i-th k-mer in *Text* is equal to SUFFIX$(a_i \mid b_i)$ for all i from 1 to n, if such a string exists.

Constraints

- The value of k will be between 1 and 10^2.
- The value of d will be between 1 and 10^3.
- The number of strings in *PairedReads* will be between 1 and 10^4.
- The length of any one pair of paired k-mers in *PairedReads* will be between 1 and 10^2.
- All k-mer strings in *PairedReads* will be DNA strings.

3M Generate All Maximal Non-Branching Paths in a Graph

Maximal Non-Branching Path Problem
Find all maximal non-branching paths in a graph.

Input: A directed graph.
Output: The collection of all maximal non-branching paths in this graph.

c3/logos/3M.png

Formatting

Input: An adjacency list representing a directed graph.
Output: A newline-separated collection of all maximal non-branching path in the graph where each maximal non-branching path is represented as a space-separated list of integer node labels.

Constraints

- The number of nodes in the graph will be between 1 and 10^3.
- The number of edges in the graph will be between 1 and 10^3.
- All nodes in the graph will be labeled with integers.

5 How Do We Compare DNA Sequences?

Dynamic Programming

5A Find the Minimum Number of Coins Needed to Make Change

The Change Problem
Find the minimum number of coins needed to make change.

Input: A non-negative integer *money* and an array *Coins* of positive integers.
Output: The minimum number of coins with denominations *Coins* that changes *money*.

Formatting

Input: A non-negative integer *money* followed by a space-separated list of positive integers *Coins*.
Output: The minimum number of coins with denominations *Coins* that changes *money*.

Constraints

- The value of *money* will be between 1 and 10^5.
- The number of positive integers in *Coins* will be between 1 and 10^1.

Test Cases

Case 1

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

Input:

```
7
1 5
```

Output:

```
3
```

Case 2

Description: This dataset makes sure that your code is correctly considering the last coin denomination in the *Coins* array. If your solution has an off-by-one indexing mistake while iterating over the *Coins* array it could be possible that the last *coin* is not considered. In this case code run on this dataset will output `2` instead of the correct answer, `1`.

Input:

```
10
1 2 3 4 5 10
```

Output:

```
1
```

Case 3

Description: This dataset makes sure that your code is correctly considering the first coin denomination in the *Coins* array. If your solution has an off-by-one indexing mistake while iterating over the *Coins* array it could be possible that the first *coin* is not considered. In this case code run on this dataset will output `2` instead of the correct answer, `1`.

Input:

```
10
10 5 4 3 2 1
```

Output:

```
1
```

Case 4

Description: This dataset checks if your code correctly returns the final value in the array in the dynamic programming approach for solving this problem. It is possible that an off-by-one indexing error (could be related to confusing 0/1 indexing) results in your code outputting the minimum number of coins needed to make change for $money-1$ instead of *money*. In this case your code will output `2` instead of the correct value of *3*.

Input:

```
11
1 5
```

Output:

```
3
```

Case 5

Description: This dataset checks to make sure you are not using a greedy algorithm to solve this problem. While a greedy algorithm in which the largest valued *coin* is used on each iteration may work in some cases, it will fail on this dataset. A greedy approach would start by using the 9 *coin* which would only allow for the use of 3 of the 1 *coin* to get to 12, resulting in an output of `4`. Using 2 of the 6 *coin* will result in exact change with only 2 coins.

Input:

```
12
9 6 1
```

Output:

```
2
```

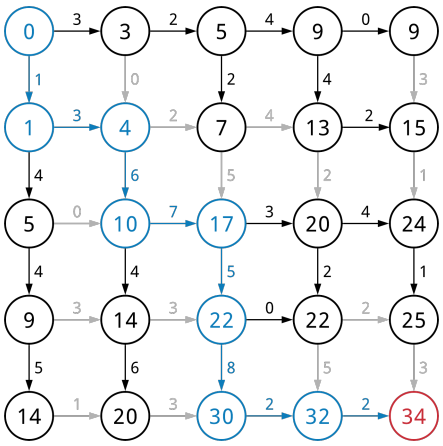
Case 6

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

5B Find the Length of a Longest Path in a Manhattan-like Grid

Length of a Longest Path in the Manhattan Tourist Problem
Find the length of a longest path in a rectangular city.

Input: Integers n and m, an $n \times (m+1)$ matrix *Down*, and an $(n+1) \times m$ matrix *Right*.
Output: The length of a longest path from source (0,0) to sink (n,m) in the $n \times m$ rectangular grid whose edges are defined by the matrices *Down* and *Right*.



Formatting

Input: Integers n and m, followed by an $n \times (m+1)$ matrix *Down*, and an $(n+1) \times m$ matrix *Right*. The two matrices are separated by the "-" symbol.
Output: The length of a longest path from source (0,0) to sink (n,m) in the $n \times m$ rectangular grid whose edges are defined by the matrices *Down* and *Right* as an integer.

Constraints

- The values of n and m will be between 1 and 10^2.

Test Cases

Case 1

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

Input:

```
4 4
1 0 2 4 3
4 6 5 2 1
4 4 5 2 1
5 6 8 5 3
-
3 2 4 0
3 2 4 2
0 7 3 3
3 3 0 2
1 3 2 2
```

Output:

```
34
```

Case 2

Description: This dataset checks that your code considers the first column of *Down* and the last row of *Right*. If there is some off-by-one error (possibly due to 0/1 indexing confusion) in updating the dynamic programming matrix it could be possible to miss these edges.

Input:

```
2 2
20 0 0
20 0 0
-
0 0
0 0
10 10
```

Output:

```
60
```

Case 3

Description: This dataset checks that your code considers the last column of Down and the first row of Right. If there is some off-by-one error (possibly due to 0/1 indexing confusion) in updating the dynamic programming matrix it could be possible to miss these edges. This dataset is very similar to test case 2.

Input:

```
2 2
0 0 20
0 0 20
-
10 10
0 0
0 0
```

Output:

```
60
```

Case 4

Description: This dataset checks that you are not using some sort of greedy approach to solving this problem. If the movement with the highest weight is chosen at each step the first entry of *Down* should be chosen. This will result in missing the higher total weight that just goes all the way right then down.

Input:

```
2 2
20 0 0
0 0 0
-
0 30
0 0
0 0
```

Output:

```
30
```

Case 5

Description: This dataset checks that your code can correctly parse and use inputs in which n and m are not the same. In all previous test datasets n and m were the same. If your output doesn't match the correct output make sure that your dynamic programming matrix has the dimensions $(n+1)\times(m+1)$. In previous datasets your code could get away with creating a dynamic programming matrix with dimensions $(n+1)\times(n+1)$ or $(m+1)\times(m+1)$, but implementations relying on those dimensions will fail this dataset.

Input:

```
5 3
20 5 0 10
0 5 10 0
10 10 0 15
0 20 20 25
30 10 5 30
-
0 30 15
10 20 10
10 10 20
20 25 30
15 35 40
15 10 25
```

Output:

```
175
```

Case 6

Description: This dataset checks that your code can correctly parse and use inputs in which n and m are not the same. This dataset is similar to test dataset 5.

Input:

```
3 5
0 5 10 0 10 10
15 0 20 20 25 30
10 5 30 15 0 20
-
0 30 15 10 20
10 10 10 20 20
25 30 15 35 40
15 10 25 15 20
```

Output:

```
180
```

Case 7

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

5C Find a Longest Common Subsequence of Two Strings

Longest Common Subsequence Problem
Find a longest common subsequence of two strings.

Input: Two strings.
Output: A longest common subsequence of these strings.

Formatting

Input: Two newline-separated strings v and w.
Output: A longest common subsequence of v and w as a string.

Constraints

- The lengths of v and w will be between 1 and 10^3.
- Both v and w will be DNA strings.

Test Cases

Case 1

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

Input:
```
GACT
ATG
```

Output:
```
AT
```

Case 2

Description: This dataset checks that you code solves the longest common *subsequence* problem instead of the related longest common *substring* problem. The longest common *substring* between **ACTG**AG and G**ACTG**G is ACGT, but the longest common *subsequence* between **ACGT**A**G** and G**ACTGG** is ACTGG.

Input:
```
ACTGAG
GACTGG
```

Output:
```
ACTGG
```

Case 3

Description: This simple dataset is used to check if your code correctly reconstructs the longest common subsequence from the backtracking matrix. Common errors include forgetting to reverse the reconstruction before returning (result: CA), terminating reconstruction too early (result: C), and starting reconstruction too late (result: A).

Input:
```
AC
AC
```

Output:
```
AC
```

Case 4

Description: This dataset checks that your code correctly considers the last character of each string. Off-by-one errors (can be caused by $0/1$ indexing errors) in indexing the input strings could result in the solution erroneously ignoring the final characters of inputs. If your code outputs an empty string it is likely that your implementation includes some off-by-one error.

Input:

```
GGGGT
CCCCT
```

Output:

```
T
```

Case 5

Description: This dataset checks that your code correctly considers the first character of each string. Off-by-one errors (can be caused by $0/1$ indexing errors) in indexing the input strings could result in the solution erroneously ignoring the first characters of inputs. If your code outputs an empty string it is likely that your implementation includes some off-by-one error.

Input:

```
TCCCC
TGGGG
```

Output:

```
T
```

Case 6

Description: This dataset checks that your code can handle inputs in which the two strings to be aligned are different lengths. If your output is incorrect make sure that your dynamic programming matrix has dimensions $(|v|+1)\times(|w|+1)$ or $(|w|+1)\times(|v|+1)$. If your code incorrectly sets the dynamic programming matrix dimensions to $(|v|+1)\times(|v|+1)$ or $(|w|+1)\times(|w|+1)$ it will not necessarily fail previous datasets since $|v|$ is the same as $|w|$ in all previous test datasets. Make sure that your implementation does not make any assumptions about the sizes of strings v and w.

Input:

```
AA
CGTGGAT
```

Output:

```
A
```

Case 7

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 6 except that in this dataset string v is longer than string w.

Input:

```
GGTGACGT
CT
```

Output:

```
CT
```

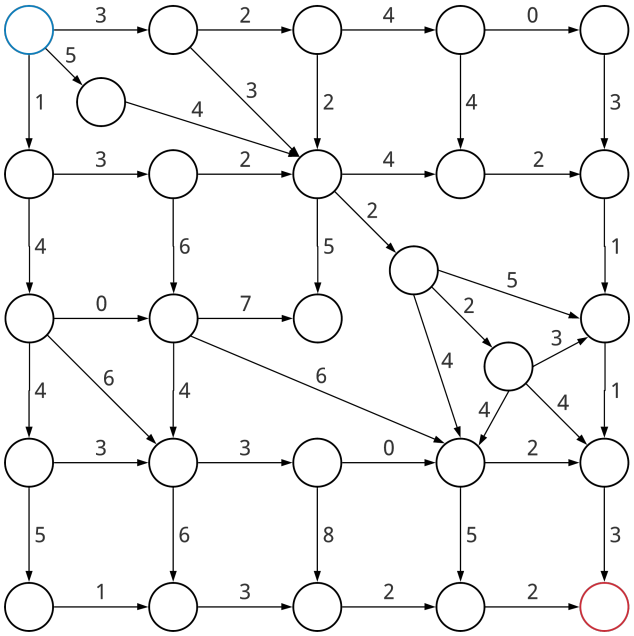
Case 8

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

5D Find a Longest Path in a DAG

Longest Path in a DAG Problem
Find the longest path between two nodes in an edge-weighted DAG.

Input: An edge-weighted directed acyclic graph (DAG) and nodes *Source* and *Sink* in this DAG.
Output: A longest path from *Source* to *Sink* in this DAG and its length.



Formatting

Input: Two space-separated integers labeling the source and sink nodes of an edge-weighted DAG, followed by an edge list representing an edge-weighted DAG.
Output: The length of a longest path in the edge-weighted DAG as an integer followed by a space-separated list of integers representing a longest path in the edge-weighted DAG (if multiple longest paths exist, you may return any one).

Constraints

- The values of *Source* and *Sink* will be between 1 and 10^2.
- The number of nodes in the DAG will be between 1 and 10^2.
- The number of edges in the DAG will be between 1 and 10^2.
- All nodes in the DAG will be labeled with integers.
- All edges in the DAG will have integer edge weights.

Test Cases

Case 1

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

Input:

```
0 4
0 1 7
0 2 4
1 4 1
2 3 2
3 4 3
```

Output:

```
9
0 2 3 4
```

Case 2

Description: This test makes sure that your code is actually finding the *longest* path in a DAG not the *shortest* path in a DAG. The *shortest* path in this DAG goes 0 1 2 3 with path length of 3 while the *longest* path goes 0 3 with a path length of 10. If your code outputs a path length of 3 it is likely that you are finding the *shortest* path instead of the *longest* path.

Input:

```
0 3
0 1 1
0 3 10
1 2 1
2 3 1
```

Output:

```
10
0 3
```

Case 3

Description: This test makes sure that your code correctly parses the input. An input line of the form x y z represents an edge from node x to node y with a weight of z. A common mistake is parsing the input so that x y z represents an edge from node x to node z with weight y. If your code outputs a path length of 4 with a path of 0 2 3 it is likely that you are making the described parsing error.

Input:

```
0 3
0 1 2
0 2 1
1 3 3
2 3 3
```

Output:

```
5
0 1 3
```

Case 4

Description: This test makes sure that your code isn't an implementation of a greedy approach to this problem. A simple greedy approach that chooses paths at branching points based on edge weights at that point is not guaranteed return the correct output for this problem. If your code outputs a path weight of 6 and a path of 0 2 3 then it's possible that you are using a greedy approach and fall into a non-optimal solution immediately by choosing the 0 2 5 edge.

Input:

```
0 3
0 1 1
0 2 5
1 3 10
2 3 1
```

Output:

```
11
0 1 3
```

Case 5

Description: This test makes sure that your code does not rely on the source node being labeled 0. This dataset is the same as test dataset 4, except that each node label is incremented by one. If your output doesn't match the correct output then your code likely assumes that the source node is labeled 0. Make sure that your implementation uses the source node label from the input instead of making any assumptions about the label of the source node.

Input:

```
1 4
1 2 1
1 3 5
2 4 10
3 4 1
```

Output:

```
11
1 2 4
```

Case 6

Description: This test makes sure that your code can correctly parse inputs in which there are double digit node labels. All previous datasets only have nodes with single digit labels, so if your code relies on node labels only having single digits it will likely fail on this dataset. If your output doesn't match the correct output make sure that your code can handle nodes that have double digit labels.

Input:

```
1 10
1 2 1
2 3 3
3 10 1
```

Output:

```
5
1 2 3 10
```

Case 7

Description: This test makes sure that your code can correctly handle inputs that only contain one edge. If your output doesn't match the correct output make sure that your implementation doesn't contain an off-by-one error that prevents the use of the only edge in the graph.

Input:

```
0 4
0 4 7
```

Output:

```
7
0 4
```

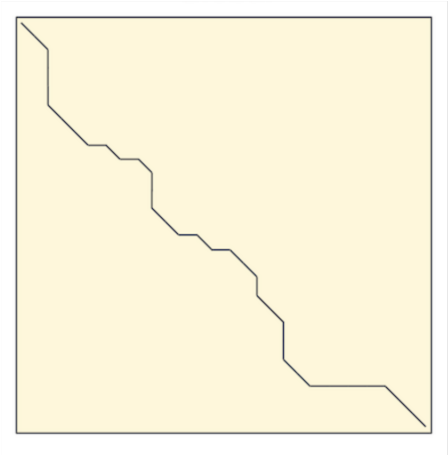
Case 8

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

5E Find a Highest-Scoring Global Alignment of Two Strings

Global Alignment Problem
Find a highest-scoring global alignment between two strings using a scoring matrix.

Input: Two DNA strings, a match reward, a mismatch penalty, and an indel penalty.
Output: A highest-scoring global alignment of these two strings (with respect to the scoring matrix and the indel penalty) and its score.



Formatting

Input: Two space-separated DNA strings v and w.
Output: The maximum global alignment score of v and w as an integer 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). Use a simple scoring function in which the match reward is equal to 1, the mismatch penalty is equal to 1, and the indel penalty is equal to 5.

Constraints

- The lengths of v and w will be between 1 and 10^4.
- 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 1 2
GAGA
GAT
```

Output:

```
-1
GAGA
GA-T
```

Case 2

Description: This test makes sure that your code correctly parses the first line of input and uses the correct penalties. The mismatch μ and indel σ penalties can easily be mistakenly swapped either when parsing the input or when actually applying the global alignment algorithm. In this case the mismatch penalty is more than twice the indel penalty. Therefore ending the alignment with two indels is better than simply aligning the mismatched final bases. If the mismatch and indel penalties were somehow switched in your code you will likely get a score of 1 and an alignment of `ACG` and `ACT`.

Input:

```
1 3 1
ACG
ACT
```

Output:

```
0
AC-G
ACT-
```

Case 3

Description: This test makes sure that the mismatch penalty is being correctly applied. A mismatch penalty of 1 means that an alignment making use of mismatched bases suffer a score decrease of 1. It can be easy to forget that penalties must be subtracted from the score, not added. Be sure that all penalties are being subtracted from score total when updating your dynamic programming matrix. Alternatively, you could negate the mismatch and indel penalties and add them to your scores. If your code outputs a score of 2 or 3 you are likely accidentally adding the penalties to your scores instead of subtracting them.

Input:

```
1 1 1
AT
AG
```

Output:

```
0
AT
AG
```

Case 4

Description: This test makes sure that your code allows for an output beginning with an indel. If your code doesn't make use of the base cases (first row and column of the dynamic programming matrix) scores then the correct score of 3 cannot be found. Be sure to correctly fill out your bases cases and consider them in your recursive cases.

Input:

```
2 5 1
TCA
CA
```

Output:

```
3
TCA
-CA
```

Case 5

Description: This test makes sure that your code can handle multiple indels in a row. If there is some indel specific error in reconstructing the alignment from your backtracking matrix you may be missing an indel at the beginning or end of the second output string. This test also makes sure that the correct score calculation is being used for global alignment. This particular dataset would have a score of 2 if fitting or local alignment were performed, but since this problem requires global alignment the preceding and trailing indels must be incorporated into the score.

Input:

```
1 10 1
TTTTCCTT
CC
```

Output:

```
-4
TTTTCCTT
----CC--
```

Case 6

Description: This test makes sure that your code can handles inputs in which the two strings differ drastically in length. Note that your reconstructed alignment may differ from the given output and is still correct as long as the "T" character in string w is aligned to one of the "T" characters in string v. If your output doesn't match the correct output make sure that your dynamic programming matrix has dimensions $(|v|+1)\times(|w|+1)$ or $(|w|+1)\times(|v|+1)$. If your code incorrectly set the dynamic programming matrix dimensions to $(|v|+1)\times(|v|+1)$ or $(|w|+1)\times(|w|+1)$ it will fail this case.

Input:

```
2 3 2
ACAGATTAG
T
```

Output:

```
-14
ACAGATTAG
-----T---
```

Case 7

Description: This test makes sure that your code can handles inputs in which the two strings differ drastically in length. This dataset is similar to test dataset 6 except in this dataset string v is much shorter than string w instead of vice-versa.

Input:

```
3 1 2
G
ACATACGATG
```

Output:

```
-15
------G----
ACATACGATAG
```

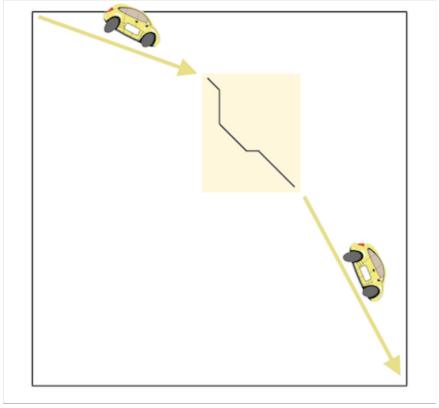
Case 8

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

5F Find a Highest-Scoring Local Alignment of Two Strings

Local Alignment Problem
Find a highest-scoring local alignment between two strings.

Input: Two DNA strings, a match reward, a mismatch penalty, and an indel penalty.
Output: A highest-scoring local alignment of these two strings (with respect to the scoring matrix and the indel penalty) and its score.



Formatting

Input: Two space-separated DNA strings v and w.
Output: The maximum local alignment score of v and w as an integer followed by a newline-separated local alignment of v and w achieving this maximum score (if multiple local alignments achieving the maximum score exist, you may return any one). Use a simple scoring function in which the match reward is equal to 1, the mismatch penalty is equal to 1, and the indel penalty is equal to 5.

Constraints

- The lengths of v and w will be between 1 and 10^4.
- 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 1 2
GAGA
GAT
```

Output:

```
2
GA
GA
```

Case 2

Description: This test makes sure that your code correctly parses the first line of input and uses the correct penalties. The mismatch (μ)and indel (σ) penalties can easily be mistakenly swapped either when parsing the input or when actually applying the local alignment algorithm. If the mismatch and indel penalties were somehow switched in your code you will likely get a score of 5 and an alignment of `AGC` and `ATC`.

Input:

```
3 3 1
AGC
ATC
```

Output:

```
4
AG-C
A-TC
```

Case 3

Description: This test makes sure that the mismatch penalties are being correctly applied. A mismatch penalty of 1 means that an alignment making use of mismatched bases suffer a score *decrease* of 1. It can be easy to forget that penalties must be subtracted from the score, not added. Be sure that all penalties are being subtracted from score total when updating your dynamic programming matrix. Alternatively, you could negate the mismatch and indel penalties and add them to your scores. If your code outputs a score of 2 or 3 you are likely accidentally adding the penalties to your scores instead of subtracting them.

Input:

```
1 1 1
AT
AG
```

Output:

```
1
A
A
```

Case 4

Description: This test makes sure that your code can correctly find the highest scoring alignment, wherever it is in the dynamic programming matrix. This test also makes sure that your code correctly backtracks for local alignments. Be sure to terminate reconstruction of the aligned strings when a "free ride" can be used back to the origin. If your code doesn't correctly terminate reconstruction it is possible that your score will be correct but your alignment will be incorrect.

Input:

```
1 1 1
TAACG
ACGTG
```

Output:

```
3
ACG
ACG
```

Case 5

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 dynamic programming matrix has dimensions $(|v|+1)\times(|w|+1)$ or $(|w|+1)\times(|v|+1)$. If your code incorrectly sets the dynamic programming matrix dimensions to $(|v|+1)\times(|v|+1)$ or $(|w|+1)\times(|w|+1)$ it will not necessarily fail previous datasets since $|v|$ is the same as $|w|$ in all previous test datasets but it will fail this one.

Input:

```
3 2 1
CAGAGATGGCCG
ACG
```

Output:

```
6
CG
CG
```

Case 6

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 5 except that in this dataset string v is shorter than string w.

Input:

```
2 3 1
CTT
AGCATAAAGCATT
```

Output:

```
5
C-TT
CATT
```

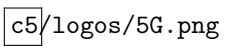
Case 7

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

5G Compute the Edit Distance Between Two Strings

Edit Distance Problem
Find the edit distance between two strings.

Input: Two strings.
Output: The edit distance between these strings.



Formatting

Input: Two space-separated strings v and w.
Output: The edit distance between v and w as an integer.

Constraints

- The lengths of v and w will be between 1 and 10^4.

Test Cases

Case 1

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

Input:

```
GAGA
GAT
```

Output:

```
2
```

Case 2

Description: This test makes sure that your code doesn't reward exact matches by adding a positive value to the edit distance. If two strings are exactly the same then their edit distance should be 0. It is easy to confuse edit distance with alignment, which could lead you to assign positive values to character matches in the dynamic programming matrix. When computing edit distance we only want to add to the edit distance when there is an indel or a mismatch. If your code outputs some multiple of 2 for this dataset it is likely that there is some mistake regarding the nature of edit distance computation. Alternatively your code could be finding *maximum* edit distance instead of *minimum* edit distance. This is an especially easy mistake to make when coming from alignment problems.

Input:

```
AC
AC
```

Output:

```
0
```

Case 3

Description: This test makes sure that your code correctly adds to the edit distance between the two strings when there are deletions or substitutions. Any sort of edit operation will add 1 to the edit distance between the two strings. If you are conflating alignment scores and edit distance it may be possible for you to come up with a negative result for this dataset. Don't forget that all singular edit operations contribute exactly 1 to the edit distance between strings; don't add different values to the edit distance for insertions, deletions, and substitutions.

Input:

```
AT
G
```

Output:

```
2
```

Case 4

Description: This test makes sure that your code correctly handles inputs in which the strings to be compared drastically differ in length. If your output doesn't match the correct output make sure that your implementation makes no assumptions about the length of the strings to be compared. Make sure that your dynamic programming matrix has dimensions $(|v|+1)\times(|w|+1)$ or $(|w|+1)\times(|v|+1)$.

Input:

```
CAGACCGAGTTAG
CGG
```

Output:

```
10
```

Case 5

Description: This test makes sure that your code correctly handles inputs in which the strings to be compared drastically differ in length. This test is similar to test dataset 4 except in this dataset string v is shorter than string w.

Input:

```
CGT
CAGACGGTGACG
```

Output:

```
9
```

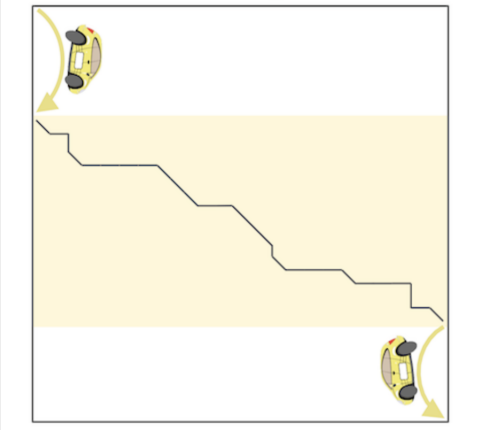
Case 6

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

5H Find a Highest-Scoring Fitting Alignment of Two Strings

Fitting Alignment Problem
Find a highest-scoring fitting alignment between two strings.

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



Formatting

Input: Two space-separated DNA strings v and w.
Output: The maximum local alignment score of v and w as an integer followed by a newline-separated local alignment of v and w achieving this maximum score (if multiple fitting alignments achieving the maximum score exist, you may return any one). Use the BLOSUM62 scoring matrix and an indel penalty of 1.

Constraints

- The lengths of v will be between 1 and 10^4
- The length of w will be between 1 and 10^3.
- 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 1 2
GAGA
GAT
```

Output:

```
1
GAG
GAT
```

Case 2

Description: This test makes sure that your dynamic programming matrix is correctly initialized. There should be no score punishment for starting at an arbitrary position in string v. Additionally, indels outside of string w should not be reported in the final alignment. If your reconstructed alignment is the whole alignment below instead of the darkened portions then double check that your alignment reconstruction implementation does not include characters that do not fall within string w. If your code outputs a score of 0 then make sure that the base cases in your dynamic programming matrix are correctly set.

Input:

```
1 1 1
CCAT
AT
```

Output:

```
2
AT
AT
```

Case 3

Description: This test makes sure that your code isn't mistakenly implementing local or global alignment instead of fitting alignment. If you are implementing local alignment you'll get a score of 1. If you are implementing global alignment you'll get a score of -2. Be careful not to confuse the different types of alignment.

Input:

```
1 5 1
CACGTC
AT
```

Output:

```
0
ACGT
A-T
```

Case 4

Description: This test makes sure that your code chooses the correct cell in the dynamic programming matrix as the final score for the fitting alignment. If your implementation outputs the score in the bottom right corner of the matrix as it would be done in global alignment you will get a score of 0. Also be sure that your code correctly backtracks from the final score cell and reconstructs the alignment.

Input:

```
1 1 1
ATCC
AT
```

Output:

```
2
AT
AT
```

Case 5

Description: This test makes sure that your code can handle inputs in which the two strings are the same size. The constraints on this problem only state that $|w| \leq |v|$, so the length of the two strings are allowed to be equal. If your output doesn't match the correct output make sure that your implementation doesn't rely on the length of string w being strictly less than the length of string v.

Input:

```
2 3 1
ACGACAGAG
CGAGAGGTT
```

Output:

```
7
CGACAGAG---
CG--AGAGGTT
```

Case 6

Description: This test makes sure that your code can correctly handles cases in which string v is significantly longer than string w. In this dataset also has a high match score and low indel and mismatch penalties, incentivizing matching characters at any cost. If your output doesn't match the correct output make sure that your implementation is correctly parsing the scoring scheme and not making any assumptions about the lengths of the strings to be aligned.

Input:

```
10 1 1
CAAGACTACTATTAG
GG
```

Output:

```
10
GACTACTATTAG
G----------G
```

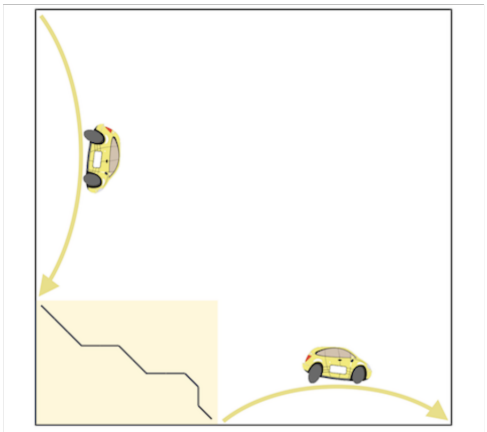
Case 7

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

5I Find a Highest-Scoring Overlap Alignment of Two Strings

Overlap Alignment Problem
Find a highest-scoring overlap alignment between two strings.

Input: Two DNA strings, a match reward, a mismatch penalty, and an indel penalty.
Output: A highest-scoring overlap alignment of these two strings (with respect to the scoring matrix and the indel penalty) and its score.



Formatting

Input: Two space-separated amino strings v and w.
Output: The maximum overlap alignment score of v and w as an integer followed by a newline-separated overlap alignment of v and w achieving this maximum score (if multiple overlap alignments achieving the maximum score exist, you may return any one). Use a simple scoring function in which the match reward is equal to 1 and both the mismatch and indel penalties are equal to 2.

Constraints

- The lengths of v and w will be between 1 and 10^3
- 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 1 2
GAGA
GAT
```

Output:

```
1
GA
GA
```

Case 2

Description: This test makes sure that your dynamic programming matrix is correctly initialized. Skipping characters at the beginning of string v should not be associated with a score penalty since the suffix of string v is the only part of interest. In other words, we can prepend any number of gaps to string w without a score penalty. For example we can write this alignment as `CCAT` and `-AT` and simply ignore the gap sequence prepended to string w and the characters they align to in string v.

Input:

```
1 1 1
CCAT
AT
```

Output:

```
2
AT
AT
```

Case 3

Description: This test makes sure that your dynamic programming matrix is correctly penalizing indels in string v. Gaps at the beginning of a suffix of string v must be penalized. In this dataset the mismatch penalty is much higher than the indel penalty so that the ideal overlap alignment has a gap in the first character of the string v suffix. If your code outputs a score of 3 then it is likely that you're mistakenly not punishing gaps at the beginning of the suffix. If your code outputs an alignment similar to `AT` and `AT` then it's likely that your alignment reconstruction is incorrectly removing characters. While characters from the beginning of string v can be freely removed this is not the case for string w. Since our alignment uses the "C" character from string w we must also include the gap it aligns to in string v.

Input:

```
1 5 1
GAT
CAT
```

Output:

```
1
-AT
CAT
```

Case 4

Description: This test makes sure that your code correctly ignores characters at the end of string w if that results in a better alignment score. In overlap alignment only the prefix of string w must be aligned. Adding the "G" character to the alignment will only hurt the score, so it is not used in an ideal overlap alignment of this dataset. If your code includes the "G" character from string w in its output then make sure that you are selecting your final score from the correct place in your dynamic programming matrix. Also check to make sure your alignment reconstruction does not add extra characters to the final alignment.

Input:

```
1 5 1
ATCACT
AT
```

Output:

```
1
ACT
A-T
```

Case 5

Description: This test makes sure that your code correctly ignores characters at the end of string w if that results in a better alignment score. In overlap alignment only the prefix of string w must be aligned. Adding the "G" character to the alignment will only hurt the score, so it is not used in an ideal overlap alignment of this dataset. If your code includes the "G" character from string w in its output then make sure that you are selecting your final score from the correct place in your dynamic programming matrix. Also check to make sure your alignment reconstruction does not add extra characters to the final alignment.

Input:

```
1 1 5
ATCACT
ATG
```

Output:

```
0
CT
AT
```

Case 6

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 dynamic programming matrix has dimensions $(|v|+1)\times(|w|+1)$ or $(|w|+1)\times(|v|+1)$. If your code incorrectly set the dynamic programming matrix dimensions to $(|v|+1)\times(|v|+1)$ or $(|w|+1)\times(|w|+1)$ it will fail this dataset.

Input:

```
3 2 1
CAGAGATGGCCG
ACG
```

Output:

```
5
-CG
ACG
```

Case 7

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 6 except that in this dataset string v is shorter than string w.

Input:

```
2 3 1
CTT
AGCATAAAGCATT
```

Output:

```
0
--CT-T
AGC-AT
```

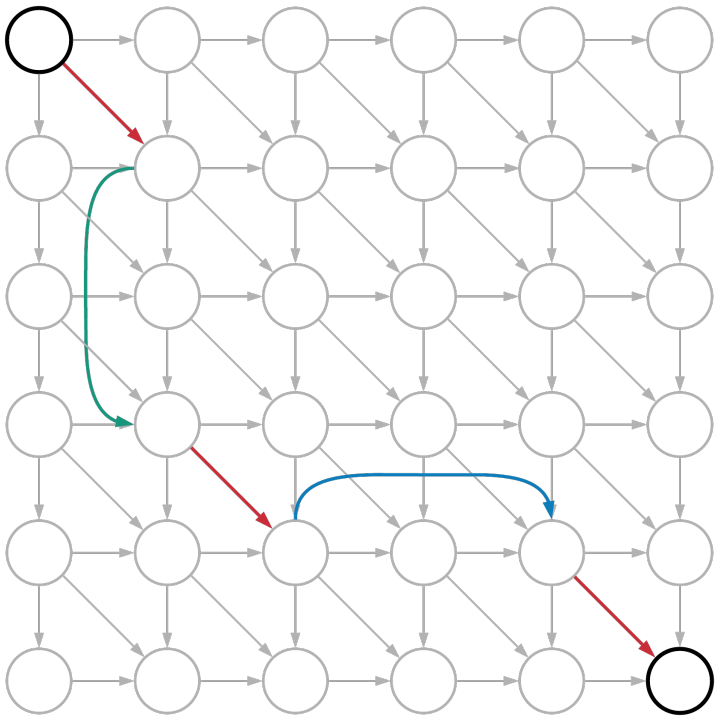
Case 8

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

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 amino acid 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 matrix and the gap opening and gap extension penalties) and its score.



Formatting

Input: Two space-separated amino strings v and w.
Output: The maximum global alignment score of v and w using 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). Use the BLOSUM62 scoring matrix, a gap opening penalty of 11, and a gap extension penalty of 1.

Constraints

- The lengths of v and w will be between 1 and 10^2
- Both v and w will be amino acid 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--
```

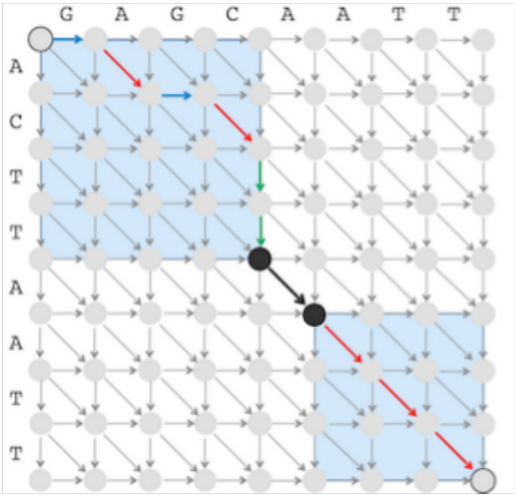
Case 10

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

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: Two space-separated amino 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. The optimal path is defined by the BLOSUM62 scoring matrix and a linear indel penalty $\sigma = 5$.

Constraints

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

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
```

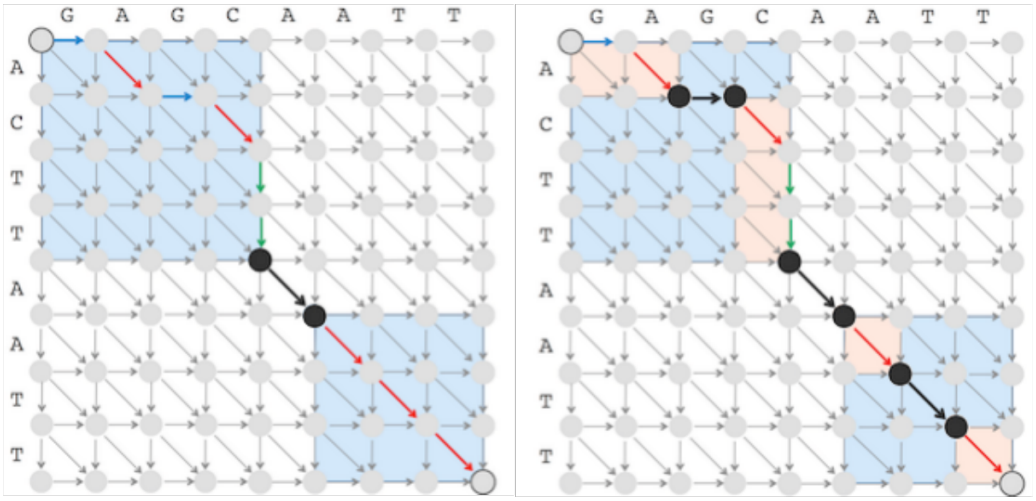
Case 8

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

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 matrix and the indel penalty) and its score.



Formatting

Input: Two space-separated amino acid strings v and w.
Output: The maximum global alignment score of v and w as an integer 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). Use the BLOSUM62 scoring matrix and an indel penalty $\sigma = 5$.

Constraints

- The lengths of v and w will be between 1 and 10^5.
- Both v and w will be amino acid strings.

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
```

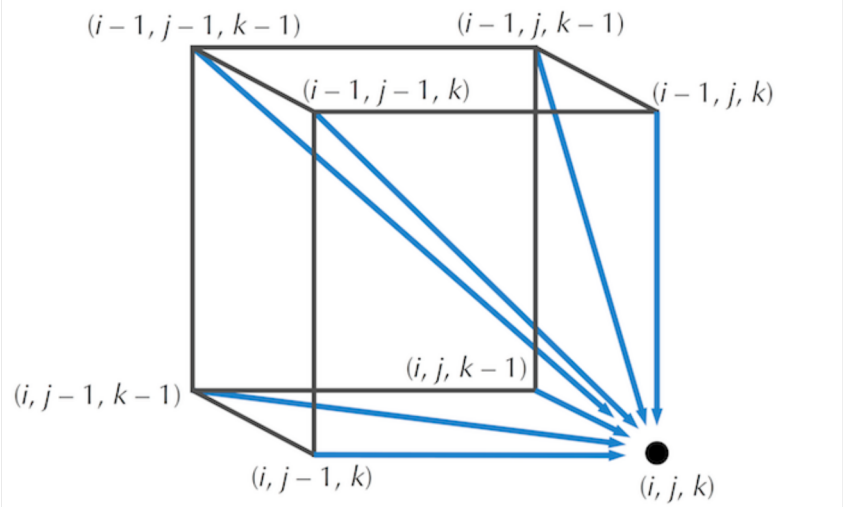
Case 8

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

5M Find a Highest-Scoring Multiple Sequence Alignment

Multiple Sequence Alignment Problem
Find a highest-scoring alignment of multiple sequences.

Input: Three DNA strings and a simple scoring function.
Output: A highest-scoring multiple alignment of these strings (with respect to the scoring matrix and indel penalty) and its score.



Formatting

Input: Three space-separated DNA strings x, y, and z.
Output: The maximum multiple sequence alignment score of x, y, and z as an integer followed by a newline-separated multiple sequence alignment of x, y, and z achieving this maximum score (if more than one multiple sequence alignments achieving the maximum score exist, you may return any one). Use a simple scoring function in which the score is equal to 1.

Constraints

- The lengths of x, y, and z will be between 1 and 10^3.
- Strings x, y, and z will be DNA strings.

Test Cases

Case 1

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

Input:

```
ATATCGG
TCCGA
ATGTACTG
```

Output:

```
3
ATATCC-G-
---TCC-GA
ATGTACTG-
```

Case 2

Description: This test makes sure that your code follows the scoring scheme of the problem. The score will only increase if there is a match between all three strings. Otherwise the score remains unchanged. In this dataset your code should not penalize the indels in reconstructed alignment for strings x and z.

Input:

```
A
AT
A
```

Output:

```
1
A-
AT
A-
```

Case 3

Description: This test makes sure that your code can accurately reconstruct the alignment when one of the aligned strings is primarily indels. When backtracking to reconstruct the alignment in this problem be sure to handle all the base cases. In regular two-string alignment you only need to worry about backtracking base cases for the very first row and the very first column of the dynamic programming matrix. In three-string alignment you also need to consider the two dimensional matrices formed by every possible pair of strings. These two dimensional matrices also need to be considered when backtracking.

Input:

```
AAAAT
CCCCT
T
```

Output:

```
1
AAAAT
CCCCT
----T
```

Case 4

Description: This test makes sure that your code correctly forces a three character match whenever possible. The reconstructed alignment is not set in stone, just make sure that the first character and last character of the three strings align with each other. Note that in the example output extra indels are introduced when not necessary (mismatching "CC" and "GG" doesn't have a penalty); the scoring scheme in this problem allows such an alignment. Make sure that your code does not unnecessarily punish extra indels.

Input:

```
AT
ACCT
AGGGGT
```

Output:

```
2
A------T
A----CCT
AGGGG--T
```

Case 5

Description: This test makes sure that your code is able to output a score of zero if there are no matching characters between the three strings. For this dataset any alignment reconstruction that includes all characters from each string is acceptable. No matter how the three strings in this dataset are aligned there is no way to obtain a non-zero score. If your output score doesn't match the correct output score make sure that your implementation doesn't disallow a score of zero when necessary.

Input:

```
GGAG
TT
CCCC
```

Output:

```
0
----GGAG
--TT----
CCCC----
```

Case 6

Description: This test makes sure that your code is able to correctly handle inputs in which all three strings are one character long. Since all the strings are the same in this dataset the output will have a score of one and the reconstructed alignment will be the same as the input. If your output doesn't match the correct output it's likely that your implementation has an error in reconstruction. If your code outputs extraneous gap characters in the reconstructed alignment there is likely an error in the termination of your reconstruction. Double check your base cases.

Input:

```
T
T
T
```

Output:

```
1
T
T
T
```

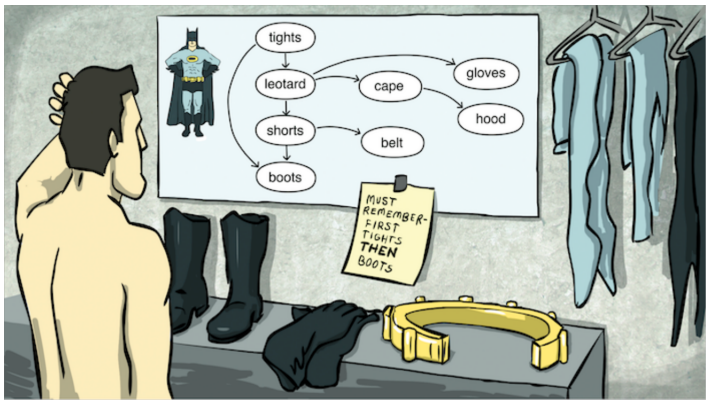
Case 7

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

5N Find a Topological Ordering of a DAG

Topological Ordering Problem
Find a topological ordering of a directed acyclic graph.

Input: A directed acyclic graph.
Output: A topological ordering of this graph.



Formatting

Input: An adjacency list representing a directed acyclic graph with nodes represented by integers.
Output: A space-separated list of integers representing a topological ordering of the DAG.

Constraints

- The number of nodes in the graph will be between 1 and 10^2.
- The number of edges in the graph will be between 1 and 10^2.
- All nodes in the graph will be labeled with integers.

Test Cases

Case 1

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 6 except that in this dataset string v is shorter than string w.

Input:

```
1:  2
2:  3
4:  2
5:  3
```

Output:

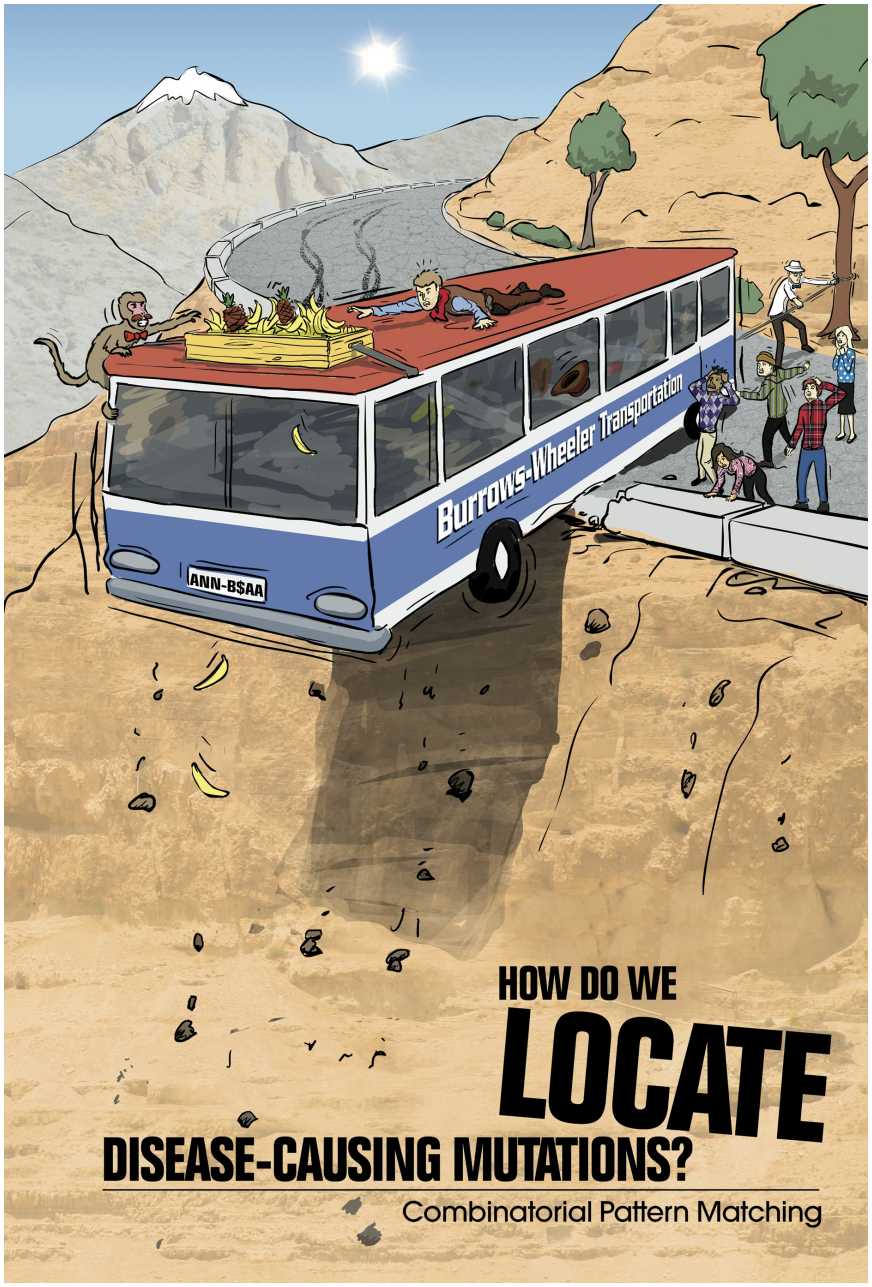
```
1 4 5 2 3
```

Case 2

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

9 How Do We Locate Disease-Causing Mutations?

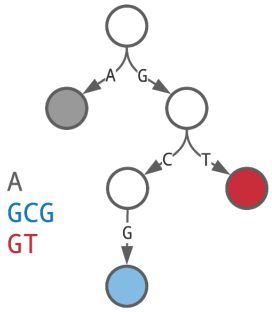
Combinatorial Pattern Matching



9A Construct a Trie from a Collection of Patterns

Trie Construction Problem
Construct a trie from a set of patterns.

Input: A collection of strings *Patterns*.
Output: TRIE(*Patterns*).



Formatting

Input: A space-separated list of strings *Patterns*.
Output: Each edge of TRIE(*Patterns*) will be newline-separated and encoded by a triple: the first two members of the triple must be the integers labeling the initial and terminal nodes of the edge, respectively; the third member of the triple must be the symbol labeling the edge.

Constraints

- The number of patterns in the string-set *Patterns* will be between 1 and 10^3.
- The length of any one pattern in *Patterns* will be between 1 and 10^3.
- No pattern is a prefix of another pattern.

Test Cases

Case 1

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

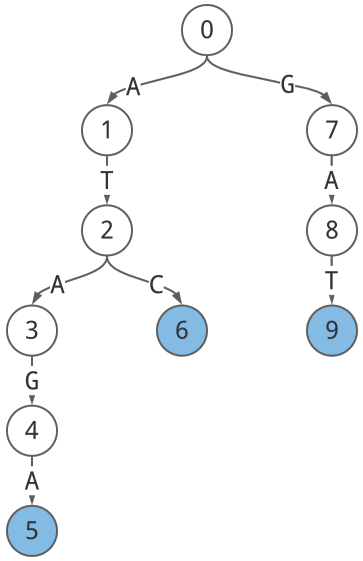
Input:

```
ATAGA ATC GAT
```

Output:

```
0 1 A
0 7 G
1 2 T
2 3 A
2 6 C
3 4 G
4 5 A
7 8 A
8 9 T
```

Figure:



Shown above is the trie containing the words `ATAGC`, `ATC`, and `GAT`. These words are outlined by the paths from the root node (labeled 0) to the leaf nodes (labeled 5, 6, and 9, colored blue).

Case 2

Description: No two patterns share the same prefix.

Input:

```
ATCG TCGA CGAT
```

Output:

```
0 1 A
0 5 T
0 9 C
1 2 T
2 3 G
3 4 C
5 6 C
6 7 G
7 8 A
9 10 G
10 11 A
11 12 T
```

Case 3

Description: All patterns share a prefix, but have distinct suffixes.

Input:

```
GAGC GAGA GAGT
```

Output:

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

Case 4

Description: Patterns have common prefixes and suffixes.

Input:

```
ATAGC ATGGC
```

Output:

```
0 1 A
1 2 T
2 3 A
3 4 G
4 5 C
2 6 G
6 7 G
7 8 C
```

Case 5

Description: Patterns comprised of repeats or palindromes.

Input:

```
ATA AGGA
```

Output:

```
0 1 A
1 2 T
2 3 A
1 4 G
4 5 G
5 6 A
```

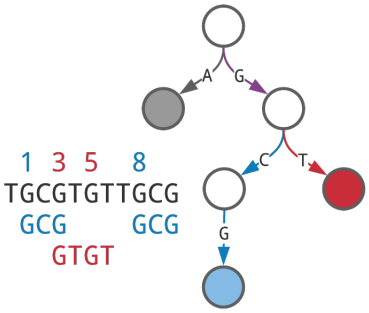
Case 6

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

9B Implement TrieMatching

Trie Matching Problem
Find all occurrence of a collection of patterns in a text.

Input: A string *Text* and a collection of (shorter) strings *Patterns*.
Output: Each string *Pattern* in *Patterns* followed by the starting positions in *Text* where *Pattern* appears as a substring.



Formatting

Input: A string *Text* and a space-separated list of strings *Patterns*.
Output: Each string *Pattern* in *Patterns* followed by the starting positions in *Text* where *Pattern* appears as a substring.

Constraints

- The length of *Text* will be between 1 and 10^4.
- The number of patterns in the string-set *Patterns* will be between 1 and 10^1.
- The length of any one pattern in *Patterns* will be between 1 and 10^4.

Test Cases

Case 1

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

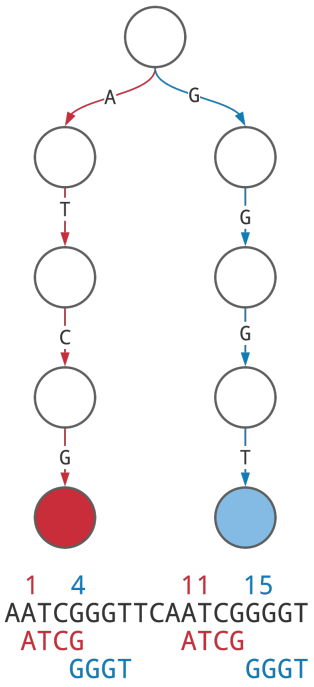
Input:

```
AATCGGGTTCAATCGGGGT
ATCG GGGT
```

Output:

```
ATCG: 1 11
GGGT: 4 14
```

Figure:



Above is the trie for `ATCG` and `GGGT` (shown in red and blue respectively). Shown below the trie is our input string *Text* as well as a representation of where our *Patterns* appear in *Text*.

Case 2

Description: There is no match for any *Pattern* in *Text*.

Input:
```
AATCGGGTTCAATCGGGGT
GGGA
```

Output:
```
GGGA:
```

Case 3

Description: A *Pattern* in *Patterns* is identical to *Text*.

Input:
```
AATC
AATC
```

Output:
```
AATC: 0
```

Case 4

Description: Patterns with repeats or palindromic sequences, overlapping occurrences of a pattern, and/or incomplete matches at *any* point in *Text*.

Input:
```
ATATATA
ATA TAT
```

Output:
```
ATA: 0 2 4
TAT: 1 3
```

Case 5

Description: Matches that only occur once (beginning and end as well).

Input:
```
GAGCAT
GAG
```

Output:
```
GAG: 0
```

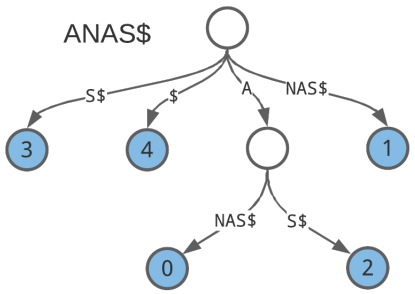
Case 6

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

9C Construct the Suffix Tree of a String

Suffix Tree Construction Problem
Construct the suffix tree of a string.

Input: A string *Text*.
Output: SUFFIXTREE(*Text*).



Formatting

Input: A string *Text* with an appended dollar-sign ("$").
Output: A space-separated list of edge labels from the constructed suffix tree (in any order).

Constraints

- The length of *Text* will be between 1 and 10^3.

Test Cases

Case 1

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

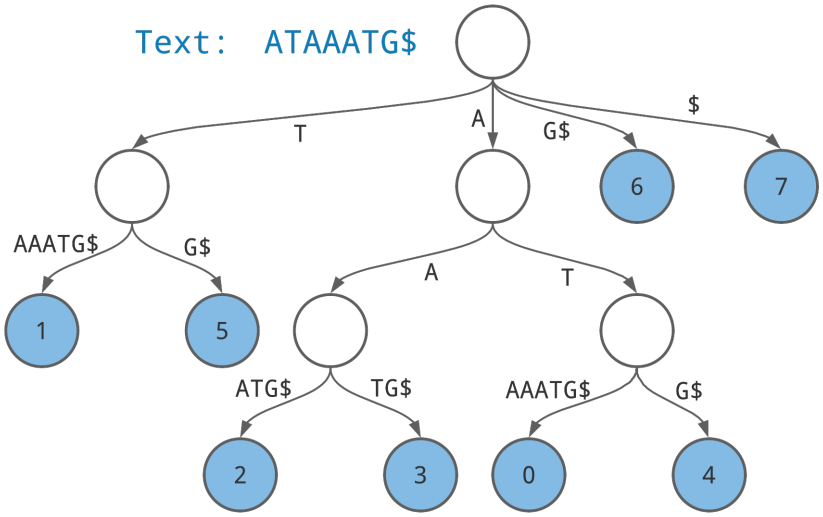
Input:

```
ATAAATG$
```

Output:

```
$ $ A A AAATG$ AAATG$ ATG G$ G$ G$ T T TG$
```

Figure:



Above is the suffix tree for the string `ATAAATG$` (notice the $ appended to the end of our input string `ATAAATG`). Each path from the root to each of the leaves (shown in blue) represents the suffix of `ATAAATG$` corresponding to the index in the leaf.

Case 2

Description: There are repeats in *Text*.

Input:
```
AATCAATC$
```

Output:
```
$ $ $ $ A AATC$ AATC$ AATC$ ATC C TC TC
```

Case 3

Description: There are no repeats in *Text*.

Input:
```
ATCG$
```

Output:
```
$ ATCG$ CG$ G$ TCG$
```

Case 4

Description: Large regions of *Text* being a single character or short tandem repeat (STR).

Input:
```
AAACA$
```

Output:
```
$ $ A AACA$ ACA$ C$ CA$
```

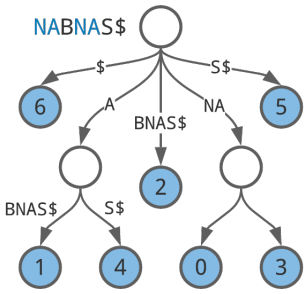
Case 5

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

9D Find the Longest Repeat in a String

Longest Repeat Problem
Find the longest repeat in a string.

Input: A string *Text*.
Output: A longest substring of *Text* that appears in *Text* more than once.



Formatting

Input: A string *Text*.
Output: The longest substring that appears more than once in *Text*.

Constraints

- The length of *Text* will be between 1 and 10^3.

Test Cases

Case 1

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

Input:
`panamabananas`

Output:
`ana`

Case 2

Description: The longest repeating sequence in *Text* overlaps with another instance of itself.

Input:
`GAGAGAG`

Output:
`GAGAG`

Case 3

Description: Multiple repeats occur with the same frequency in *Text* and are the same length.

Input:
`AGAGCTCT`

Output:
`AG` *or* `CT` (*you will not be penalized for having one over the other, but make sure you only output one*).

Case 4

Description: Repeats that occur at the beginning and end of *Text*.

Input:
`AAGCTGAA`

Output:
`AA`

Case 5

Description: There are no repeats in *Text*.

Input:

```
ABCDEFG
```

Output:

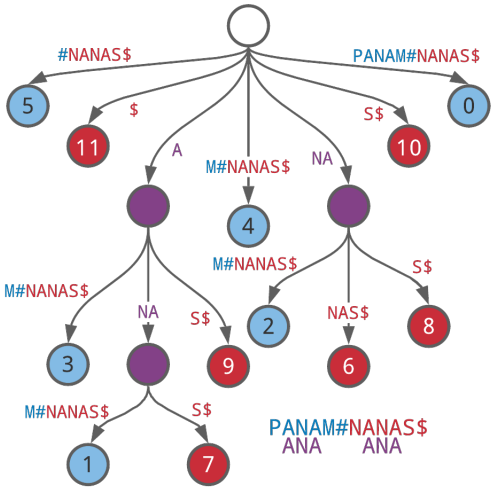
Case 6

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

9E Find the Longest Substring Shared by Two Strings

Longest Shared Substring Problem
Find the longest substring shared by two strings.

Input: Strings $Text_1$ and $Text_2$.
Output: The longest substring that occurs in both $Text_1$ and $Text_2$.



Formatting

Input: A pair of strings $Text_1$ and $Text_2$
Output: The longest substring that occurs in both $Text_1$ and $Text_2$

Constraints

- The lengths of $Text_1$ and $Text_2$ will be between 1 and 10^3.

Test Cases

Case 1

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

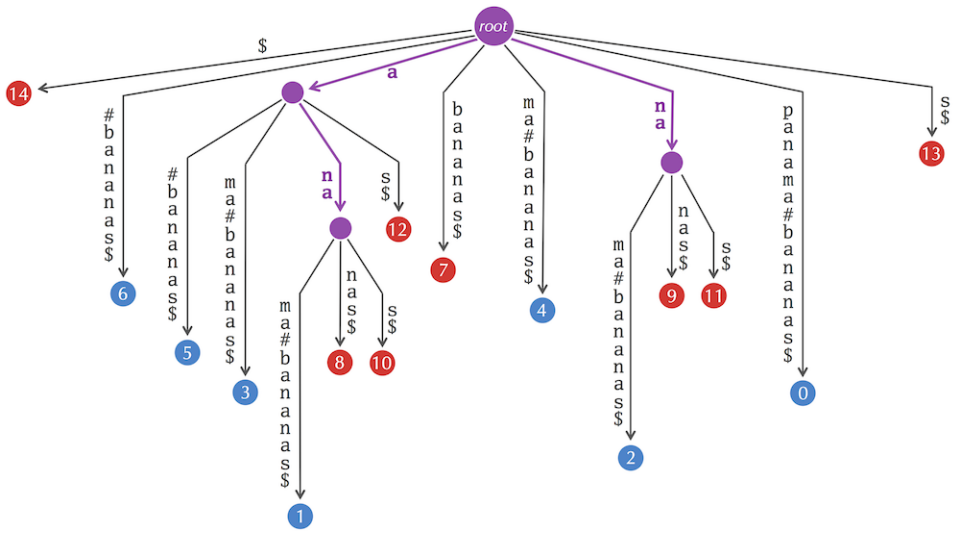
Input:

```
panama
bananas
```

Output:

```
ana
```

Figure:



Shown above is the suffix tree of the string `panama#bananas$`. Blue and red leaves represent suffixes that start in `panama` and `bananas`, respectively. An internal node is colored purple if it has both blue and red descendants. Each purple node is a shared substring of `panama` and `bananas`. The longest shared substring (purple node) is `ana`.

Case 2

Description: $Text_1$ and $Text_2$ have no common substring.

Input:
GAGA
CTCT

Output:

Case 3

Description: $Text_1$ and $Text_2$ only share 1-mers.

Input:
GAGT
GGCT

Output:
C *or* G *or* T (*you will not be penalized for having one over the other, but make sure you only output one*).

Case 4

Description: $Text_1 = Text_2$.

Input:
GAGCAT
GAGCAT

Output:
GAGCAT

Case 5

Description: The suffix of $Text_1$ and the prefix of $Text_2$ are the same.

Input:
GAGCAT
CATAGA

Output:
CAT

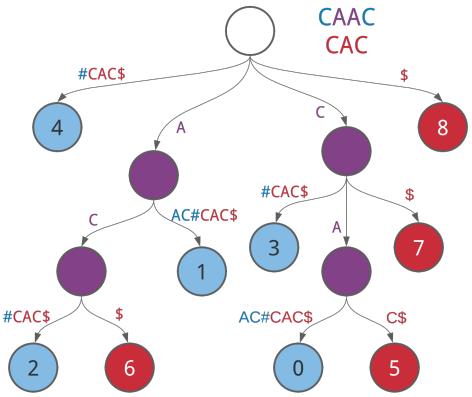
Case 6

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

9F Find the Shortest Non-Shared Substring of Two Strings

Shortest Non-Shared Substring Problem
Find the shortest substring of one string that does not appear in another string.

Input: Strings $Text_1$ and $Text_2$.
Output: The shortest substring of $Text_1$ that does not appear in $Text_2$.



Formatting

Input: A pair of strings $Text_1$ and $Text_2$.
Output: The shortest substring of $Text_1$ that does not appear in $Text_2$.

Constraints

- The lengths of $Text_1$ and $Text_2$ will be between 1 and 10^3.

Test Cases

Case 1

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

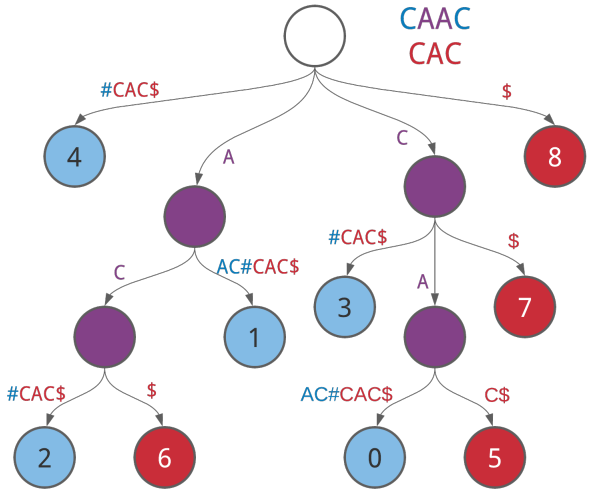
Input:

```
CAAC
CAC
```

Output:

```
AA
```

Figure:



Case 2

Description: $Text_1$ and $Text_2$ are identical.

Input:
`GAGCAT`
`GAGCAT`

Output:

Case 3

Description: $Text_1$ and $Text_2$ only differ by one character.

Input:
`GAGT`
`GAGC`

Output:
`T`

Case 4

Description: $Text_1$ and $Text_2$ are completely different (no shared characters).

Input:
`GG`
`CT`

Output:
`C` or `G` or `T` (*you will not be penalized for having one over the other, but make sure you only output one*).

Case 5

Description: $Text_1$'s prefix is the same as $Text_2$'s suffix, or vice versa.

Input:
`CGAGCATA`
`ATACGAGC`

Output:
`AC` or `CA` (*you will not be penalized for having one over the other, but make sure you only output one*).

Case 6

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

9G Construct the Suffix Array of a String

Suffix Array Construction Problem
Construct the suffix array of a string.

Input: A string *Text*.
Output: SUFFIXARRAY(*Text*).

```
7  $
1  ANANAS$
3  ANAS$
5  AS$
0  BANANAS$
2  NANAS$
4  NAS$
6  S$
```

Formatting

Input: A string *Text*.
Output: A space-separated list of integers corresponding to SUFFIXARRAY(*Text*).

Constraints

- The length of *Text* will be between 1 and 10^3.

Test Cases

Case 1

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

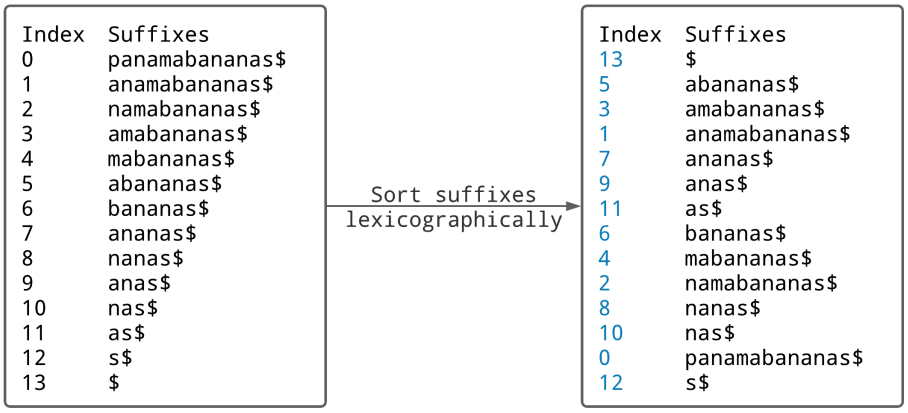
Input:

```
panamabananas
```

Output:

```
13 5 3 1 7 9 11 6 4 2 8 10 0 12
```

Figure:



Shown above is a general (and inefficient) construction of the suffix array of the input string `panamabananas`. We first generate all suffixes of *Text* before sorting the suffixes lexicographically and outputting the indices representing the sorted suffixes as the complete suffix array of *Text*.

Case 2

Description: There are repeats in *Text*.

Input:
```
AATCAATC
```

Output:
```
8 4 0 5 1 6 2 7 3
```

Case 3

Description: There are no repeats in *Text*.

Input:
```
ATCG
```

Output:
```
4 0 2 3 1
```

Case 4

Description: Large regions of *Text* being a single character or short tandem repeat (STR).

Input:
```
AAACA
```

Output:
```
5 4 0 1 2 3
```

Case 5

Description: Many different characters in one pattern.

Input:
```
ABCFED
```

Output:
```
6 0 1 2 5 4 3
```

Case 6

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

9H Pattern Matching with the Suffix Array

Multiple Pattern Matching with the Suffix Array
Use the suffix array of a string to find all occurrences of a collection of Patterns.

Input: A string *Text* and a collection *Patterns* containing (shorter) strings.
Output: All starting positions in *Text* where a string from *Patterns* appears as a substring.

```
7  $
1  ANANAS$
3  ANAS$
5  AS$          ANA: 1 3
0  BANANAS$     BAN: 0
2  NANAS$
4  NAS$
6  S$
```

Formatting

Input: A string *Text* and a space-separated list of strings *Patterns*
Output: A newline-separated list of strings from *Patterns*. Each *Pattern* in *Patterns* is followed by a colon (":") and a space-separated list of starting indices in *Text* where *Pattern* appears as a substring.

Constraints

- The length of *Text* will be between 1 and 10^4.
- The number of patterns in the string-set *Patterns* will be between 1 and 10^1.
- The length of any one pattern in *Patterns* will be between 1 and 10^4.

Test Cases

Case 1

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

Input:

```
AATCGGGTTCAATCGGGGT
ATCG GGGT
```

Output:

```
ATCG: 1 11
GGGT: 4 15
```

Figure:

Burrows-Wheeler Matrix	Suffix Array
$AATCGGGTTCAATCGGGGT	19
AATCGGGGT$AATCGGGTTC	10
AATCGGGTTCAATCGGGGT$	0
ATCGGGGT$AATCGGGTTCA	11
ATCGGGTTCAATCGGGGT$A	1
CAATCGGGGT$AATCGGGTT	9
CGGGGT$AATCGGGTTCAAT	13
CGGGTTCAATCGGGGT$AAT	3
GGGGT$AATCGGGTTCAATC	14
GGGT$AATCGGGTTCAATCG	15
GGGTTCAATCGGGGT$AATC	4
GGT$AATCGGGTTCAATCGG	16
GGTTCAATCGGGGT$AATCG	5
GT$AATCGGGTTCAATCGGG	17
GTTCAATCGGGGT$AATCGG	6
T$AATCGGGTTCAATCGGGG	18
TCAATCGGGGT$AATCGGGT	8
TCGGGGT$AATCGGGTTCAA	12
TCGGGTTCAATCGGGGT$AA	2
TTCAATCGGGGT$AATCGGG	7

The complete Burrows-Wheeler matrix shown above can be inferred using only the Burrows-Wheeler transform of this string and the Last-To-First property of the Burrows-Wheeler transform. We then search for our query strings, `ATCG` and `GGGT`, as prefixes in the rows of our matrix. Finally, we can use the suffix array of the database string to locate the positions of query matches.

Case 2

Description: There are no matches in *Text* to any pattern in *Patterns*.

Input:

```
ATATATATAT
GT AGCT TAA AAT AATAT
```

Output:

```
GT:
AGCT:
TAA:
AAT:
AATAT:
```

Case 3

Description: *Text* contains overlapping occurrences of *Patterns*.

Input:

```
bananas
ana as
```

Output:

```
ana:  1 3
as:  5
```

Case 4

Description: Large regions of *Text* being a single character or short tandem repeat (STR).

Input:

```
AAACAA
AA
```

Output:

```
AA: 0 1 4
```

Case 5

Description: *Text* is palindromic or has substrings that are palindromic.

Input:

```
GAGCAT
GA AG
```

Output:

```
GA: 0
AG: 1
```

Case 6

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

9I Construct the Burrows-Wheeler Transform of a String

Burrows-Wheeler Transform Construction Problem
Construct the Burrows-Wheeler transform of a string.

Input: A string *Text*.
Output: BWT(*Text*).

$BANANAS
ANANAS$B
ANAS$BAN
AS$BANAN
BANANAS$
NANAS$BA
NAS$BANA
S$BANANA

Formatting

Input: A string *Text*.
Output: A string *Transform* representing BWT(*Text*).

Constraints

- The length of *Text* will be between 1 and 10^3.

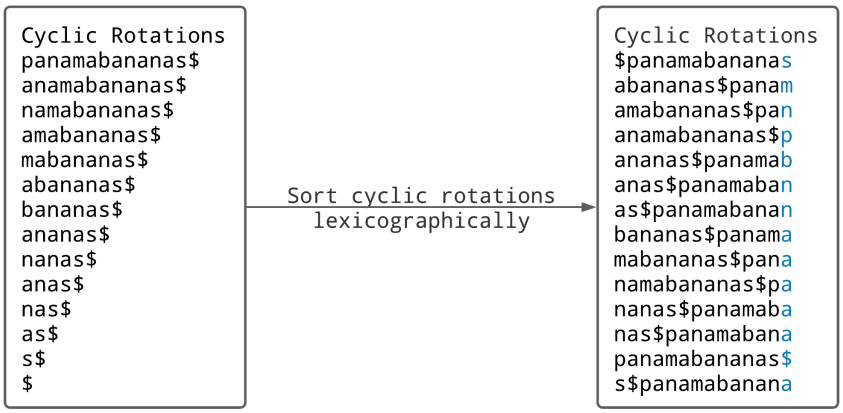
Test Cases

Case 1

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

Output:
`smnpbnnaaaaa$a`

Figure:



Shown above is a general (and inefficient) construction process for the Burrows-Wheeler Transform for the string `panamabananas`. We first generate all cyclic rotations of *Text* before sorting them lexicographically to build a Burrows-Wheeler matrix. Lastly, we output the last column of the Burrows-Wheeler matrix as the Burrows-Wheeler Transform of *Text*. This process is very similar to the process of generating a suffix array.

Case 2

Description: There are repeats in *Text*.

Input:
`AATCAATC`

Output:
`CC$AATTAA`

Case 3

Description: *Text* is made up of only one character.

Input:
`AAAAAAAAAA$`

Output:
`AAAAAAAAAA$`

Case 4

Description: *Text* is palindromic or has substrings that are palindromic.

Input:
`GAGCAT$`

Output:
`TGCG$AA`

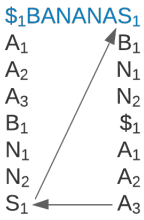
Case 5

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

9J Reconstruct a String from its Burrows-Wheeler Transform

Inverse Burrows-Wheeler Transform Problem
Reconstruct a string from its Burrows-Wheeler transform.

Input: A string *Transform* (with a single "$" symbol).
Output: The string *Text* such that BWT(*Text*)=*Transform*



Formatting

Input: A string *Transform*
Output: A string *Text* such that BWT(*Text*)=*Transform*.

Constraints

- The length of *Transform* will be between 1 and 10^3.

Test Cases

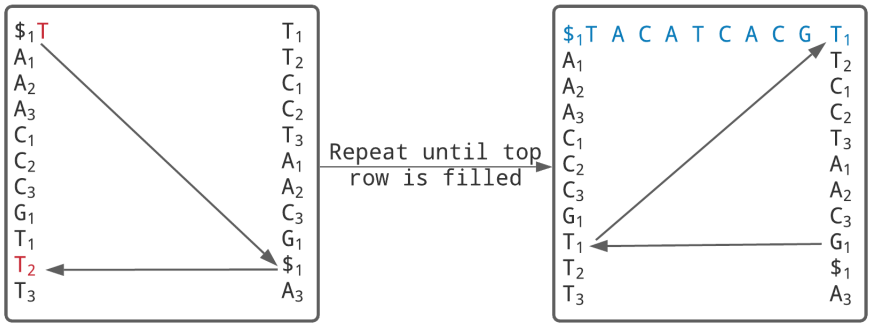
Case 1

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

Input:
`TTCCTAACG$A`

Output:
`TACATCACGT$`

Figure:



Above is a general overview of the BWT inversion process. `TTCCTAACG$A` is BWT(*Text*), and we repeat the first-last traversal process until we have "filled" the top row of the BWT matrix. Lastly, we rotate the top row until the `$` is at the end of the string to obtain `TACATCACGT$`.

Case 2

Description: There are no repeat characters in *Text*.

Input:
`T$ACG`

Output:
`ACGT$`

Case 3

Description: *Text* is made up of only one character.

Input:
`AAAAAAAAAA$`

Output:
`AAAAAAAAAA$`

Case 4

Description: *Text* is palindromic or has substrings that are palindromic.

Input:
`TGCG$AA`

Output:
`GAGCAT$`

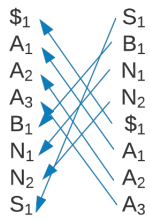
Case 5

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

9K Generate the Last-to-First Mapping of a String

Last-to-First Mapping Problem
Construct a Last-to-First Array of a Burrows-Wheeler Transform.

Input: A string *Transform* representing the Burrows-Wheeler transform of an unknown string *Text*.
Output: An array *LastToFirst* that provides the following information: given a symbol at position i in *LastColumn* of the Burrows-Wheeler matrix of *Text*, LASTTOFIRST(i) reveals its position in *FirstColumn*.



Formatting

Input: A string *Transform* representing the Burrows-Wheeler transform of an unknown string *Text*.
Output: A space-separated list of integers representing the array *LastToFirst*.

Constraints

- The length of *Transform* will be between 1 and 10^3.
- The value of i will be between 1 and 10^3.

Test Cases

Case 1

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

Input:
```
smnpbnaaaaa$a
```

Output:
```
13 8 9 12 7 10 11 1 2 3 4 5 0 6
```

Case 2

Description: There are no repeat characters in *Text*.

Input:
```
T$ACG
```

Output:
```
4 0 1 2 3
```

Case 3

Description: *Text* is made up of only one character.

Input:
```
AAAAAAAAAA$
```

Output:
```
1 2 3 4 5 6 7 8 9 10 0
```

Case 4

Description: *Text* is palindromic or has substrings that are palindromic.

Input:
```
TGCG$AA
```

Output:
```
5 0 3 2 6 1 4
```

Case 5

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

9L Implement BWMatching

Implement BWMatching
Count all occurrences of a collection of patterns in a text.

Input: A string BWT(*Text*), followed by a collection of strings *Patterns*.
Output: A list of integers, where the *i*-th integer corresponds to the number of substring matches of the *i*-th member of *Patterns* in *Text*.

```
                $BANANAS
                ANANAS$B
  2  1          ANAS$BAN
                AS$BANAN
BANANAS$        BANANAS$
ANA  AS         NANAS$BA
 ANA            NAS$BANA
                S$BANANA
```

Formatting

Input: A string BWT(*Text*) and a space-separated list of strings *Patterns*.
Output: A space-separated list of integers, where the *i*-th integer corresponds to the number of substring matches of the *i*-th member of *Patterns* in *Text*.

Constraints

- The length of BWT(*Text*) will be between 1 and 10^4.
- The number of strings in *Patterns* will be between 1 and 10^1.
- The length of any given string in *Patterns* will be between 1 and 10^4.

Test Cases

Case 1

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

Input:

```
TCCTCTATGAGATCCTATTCTATGAAACCTTCA$GACCAAAATTCTCCGGC
CCT CAC GAG CAG ATC
```

Output:

```
2 1 1 0 1
```

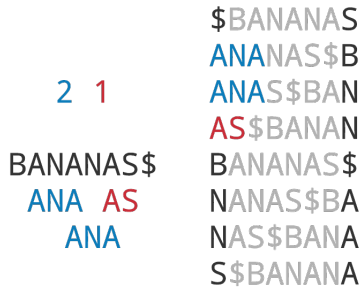
Case 2

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

9M Implement BetterBWMatching

Implement BetterBWMatching
Count all occurrences of a collection of patterns in a text.

Input: A string BWT(*Text*), followed by a collection of strings *Patterns*.
Output: A list of integers, where the *i*-th integer corresponds to the number of substring matches of the *i*-th member of *Patterns* in *Text*.



Formatting

Input: A string BWT(*Text*) and a space-separated list of strings *Patterns*.
Output: A space-separated list of integers, where the *i*-th integer corresponds to the number of substring matches of the *i*-th member of *Patterns* in *Text*.

Constraints

- The length of BWT(*Text*) will be between 1 and 10^4.
- The number of strings in *Patterns* will be between 1 and 10^1.
- The length of any given string in *Patterns* will be between 1 and 10^4.

Test Cases

Case 1

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

Input:

```
GGCGCCGC$TAGTCACACACGCCGTA
ACC CCG CAG
```

Output:

```
1 2 1
```

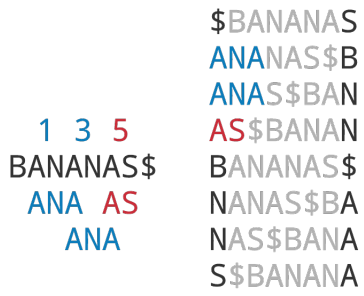
Case 2

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

9N Find All Occurrences of a Collection of Patterns in a String

Multiple Pattern Matching Problem
Find all occurrences of a collection of patterns in a text.

Input: A string *Text* and a collection *Patterns* containing (shorter) strings.
Output: All starting positions in *Text* where a string from *Patterns* appears as a substring.



Formatting

Input: A string *Text* and a space-separated list of strings *Patterns*
Output: A newline-separated list of strings from *Patterns*. Each *Pattern* in *Patterns* is followed by a colon (":") and a space-separated list of starting indices in *Text* where *Pattern* appears as a substring.

Constraints

- The length of *Text* will be between 1 and 10^4.
- The number of patterns in the string-set *Patterns* will be between 1 and 10^1.
- The length of any one pattern in *Patterns* will be between 1 and 10^4.

Test Cases

Case 1

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

Input:

```
AATCGGGTTCAATCGGGGT
ATCG GGGT
```

Output:

```
ATCG: 1 11
GGGT: 4 15
```

Figure:

Burrows-Wheeler Matrix	Suffix Array
$AATCGGGTTCAATCGGGGT	19
AATCGGGGT$AATCGGGTTC	10
AATCGGGTTCAATCGGGGT$	0
ATCGGGGT$AATCGGGTTCA	11
ATCGGGTTCAATCGGGGT$A	1
CAATCGGGGT$AATCGGGTT	9
CGGGGT$AATCGGGTTCAAT	13
CGGGTTCAATCGGGGT$AAT	3
GGGGT$AATCGGGTTCAATC	14
GGGT$AATCGGGTTCAATCG	15
GGGTTCAATCGGGGT$AATC	4
GGT$AATCGGGTTCAATCGG	16
GGTTCAATCGGGGT$AATCG	5
GT$AATCGGGTTCAATCGGG	17
GTTCAATCGGGGT$AATCGG	6
T$AATCGGGTTCAATCGGGG	18
TCAATCGGGGT$AATCGGGT	8
TCGGGGT$AATCGGGTTCAA	12
TCGGGTTCAATCGGGGT$AA	2
TTCAATCGGGGT$AATCGGG	7

The complete Burrows-Wheeler matrix shown above can be inferred using only the Burrows-Wheeler transform of this string and the Last-To-First property of the Burrows-Wheeler transform. We then search for our query strings, `ATCG` and `GGGT`, as prefixes in the rows of our matrix. Finally, we can use the suffix array of the database string to locate the positions of query matches.

Case 2

Description: There are no matches in *Text* to any pattern in *Patterns*.

Input:

```
ATATATATAT
GT AGCT TAA AAT AATAT
```

Output:

```
GT:
AGCT:
TAA:
AAT:
AATAT:
```

Case 3

Description: *Text* contains overlapping occurrences of *Patterns*.

Input:

```
bananas
ana as
```

Output:

```
ana:  1 3
as:  5
```

Case 4

Description: Large regions of *Text* being a single character or short tandem repeat (STR).

Input:

```
AAACAA
AA
```

Output:

```
AA: 0 1 4
```

Case 5

Description: *Text* is palindromic or has substrings that are palindromic.

Input:

```
GAGCAT
GA AG
```

Output:

```
GA: 0
AG: 1
```

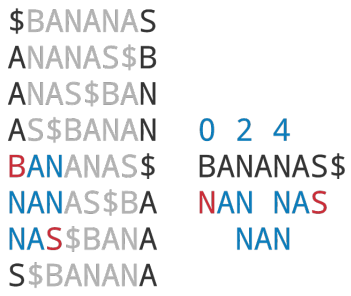
Case 6

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

9O Find All Approximate Occurrences of a Collection of Patterns in a String

Multiple Approximate Pattern Matching Problem
Find all approximate occurrences of a collection of patterns in a text.

Input: A string *Text*, and a collection of strings *Patterns*, and an integer *d*.
Output: All positions in *Text* where a string from *Patterns* appears as a substring with at most *d* mismatches.



Formatting

Input: A string *Text*, a space-separated list of strings *Patterns*, and an integer *d*
Output: A newline-separated list of strings from *Patterns*. Each *Pattern* in *Patterns* is followed by a colon (":") and a space-separated list of starting indices in *Text* where *Pattern* appears as a substring with at most *d* mismatches.

Constraints

- The length of *Text* will be between 1 and 10^4.
- The number of patterns in the string-set *Patterns* will be between 1 and 10^3.
- The length of any one pattern in *Patterns* will be between 1 and 10^2.

Test Cases

Case 1

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

Input:

```
ACATGCTACTTT
ATT GCC GCTA TATT
1
```

Output:

```
ATT: 2 7 8 9
GCC: 4
GCTA: 4
TATT: 6
```

Case 2

Description: *Patterns* contains partial and complete matches.

Input:

```
ACGT
GG AC
1
```

Output:

```
GG: 1 2
AC: 0
```

Case 3

Description: *Patterns* contains no matches.

Input:

```
GGGGG
TT AA
1
```

Output:

```
TT:
AA:
```

Case 4

Description: *Patterns* contains no matches, but has exactly *d* mismatches.

Input:

```
GG
TT AA
2
```

Output:

```
TT: 0
AA: 0
```

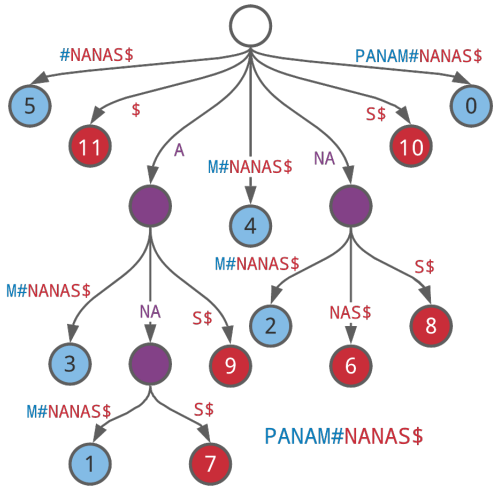
Case 5

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

9P Implement TreeColoring

Tree Coloring Problem
Color the internal nodes of a suffix tree given colors of the leaves.

Input: A tree with leaves colored red or blue.
Output: Coloring of all internal nodes of the tree such that a node is colored red if all of its descendants are red, blue if all of its descendants are blue, and purple if it has both red and blue descendants.



Formatting

Input: An newline-separated adjacency list, a delimiting character "-", and a newline-separated list of color labels for leaf nodes.
Output: A newline-separated list of nodes and their color labels in the following form: *node label color label*.

Constraints

- The number of nodes in the adjacency list will be between 1 and 10^2.
- The number of edges in the adjacency list will be between 1 and 10^2.

Test Cases

Case 1

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

Input:

```
0:
1:
2:  0 1
3:
4:
5:  2 3
6:
7:  4 5 6
-
0 red
1 red
3 blue
4 blue
6 red
```

Output:

```
0 red
1 red
2 red
3 blue
4 blue
5 purple
6 red
7 purple
```

Case 2

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

9Q Construct the Partial Suffix Array of a String

Partial Suffix Array Construction Problem
Construct the partial suffix array of a string.

Input: A string *Text* and a positive integer k.
Output: SUFFIXARRAY$_k$(*Text*), in the form of a list of ordered pairs (i, SUFFIXARRAY(i)) for all nonempty entries in the partial suffix array.

```
7  $
1  ANANAS$
3  ANAS$
5  AS$
0  BANANAS$
2  NANAS$
4  NAS$
6  S$
```

Formatting

Input: A string *Text* and a positive integer k.
Output: A newline-separated list of space-separated ordered pairs (i, SUFFIXARRAY(i)) for all nonempty entries in SUFFIXARRAY$_k$(*Text*).

Constraints

- The length of *Text* will be between 1 and 10^5.
- The integer k will be between 1 and 10^1.

Test Cases

Case 1

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

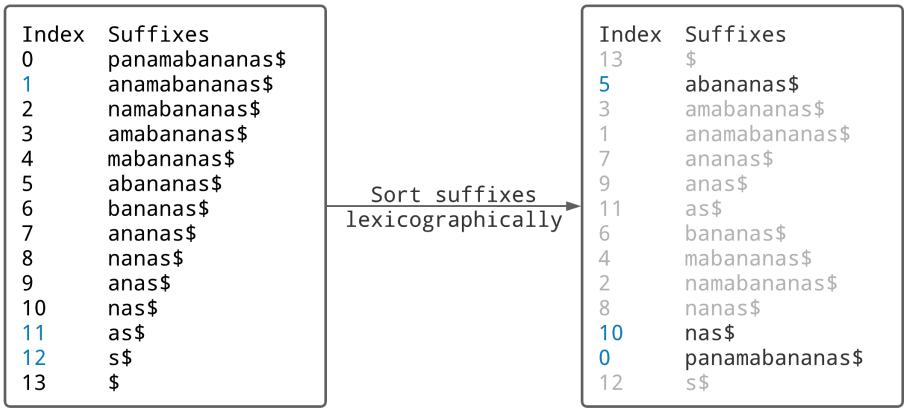
Input:

```
panamabananas$
5
```

Output:

```
1 5
11 10
12 0
```

Figure:



Shown above is a general (and inefficient) construction of the partial suffix array of the input string `panamabananas` with $k = 5$. We first generate all suffixes of *Text* before sorting the suffixes lexicographically and outputting the indices representing the sorted suffixes as the complete suffix array of *Text*. Finally, we output only the indices divisible by $k = 5$.

Case 2

Description: There are repeats in *Text*.

Input:

```
AATCAATC$
4
```

Output:

```
0 8
1 4
2 0
```

Case 3

Description: There are no repeats in *Text*.

Input:

```
ATCG$
3
```

Output:

```
1 0
3 3
```

Case 4

Description: Large regions of *Text* being a single character or short tandem repeat (STR).

Input:

```
AAACA$
5
```

Output:

```
0 5
2 0
```

Case 5

Description: Many different characters in one pattern.

Input:

```
ABCFED$
3
```

Output:

```
0 6
1 0
6 3
```

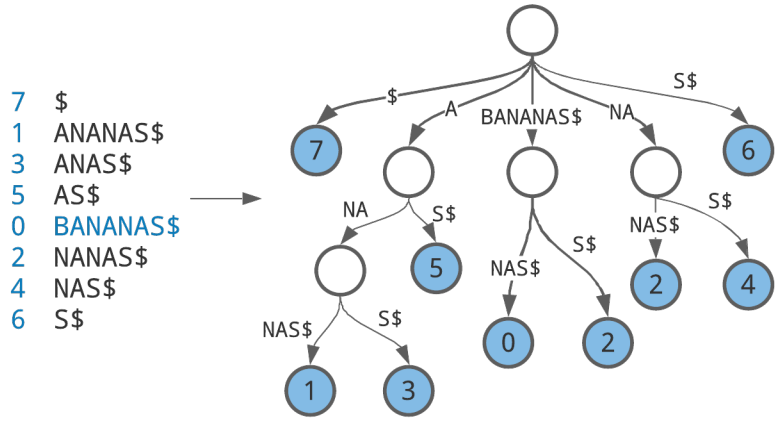
Case 6

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

9R Construct a Suffix Tree from a Suffix Array

Suffix Tree Construction from Suffix Array Problem
Construct a suffix tree from the suffix array and LCP array of a string.

Input: A string *Text*, SUFFIXARRAY(*Text*), LCP(*Text*).
Output: The strings labeling the edges of SUFFIXTREE(*Text*), in any order.



Formatting

Input: A string *Text*, followed by SUFFIXARRAY(*Text*), followed by *LCP*(*Text*).
Output: A space-separated list of edge labels from the constructed suffix tree (in any order).

Constraints

- The length of *Text* will be between 1 and 10^3.
- The length of SUFFIXARRAY(*Text*) will be between 1 and 10^3.
- The length of LCP(*Text*) will be between 1 and 10^3.

Test Cases

Case 1

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

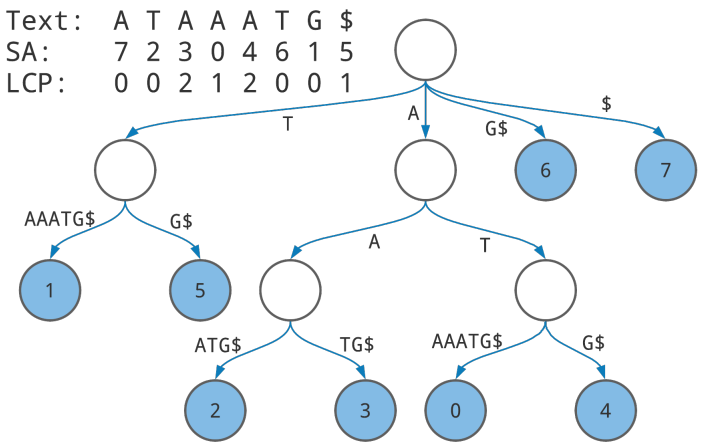
Input:

```
ATAAATG$
7 2 3 0 4 6 1 5
0 0 2 1 2 0 0 1
```

Output:

```
$ $ A A AAATG$ AAATG$ ATG G$ G$ G$ T T TG$
```

Figure:



Above is the suffix tree for the string `ATAAATG$` (notice the $ appended to the end of our input string `ATAAATG`). Each path from the root to each of the leaves (shown in blue) represents the suffix of `ATAAATG$` corresponding to the index in the leaf.

Case 2

Description: There are repeats in *Text*.

Input:

```
AATCAATC$
8 4 0 5 1 7 3 6 2
0 0 4 1 3 0 1 0 2
```

Output:

```
$ $ $ $ A AATC$ AATC$ AATC$ ATC C TC TC
```

Case 3

Description: There are no repeats in *Text*.

Input:

```
ATCG$
4 0 2 3 1
0 0 0 0 0
```

Output:

```
$ ATCG$ CG$ G$ TCG$
```

Case 4

Description: Large regions of *Text* being a single character or short tandem repeat (STR).

Input:

```
AAACA$
5 4 0 1 2 3
0 0 1 2 1 0
```

Output:

```
$ $ A A ACA$ CA$ CA$ CA$
```

Case 5

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