COSC 2P03 – Advanced Data Structures
Fall 2020
Assignment #1

Due Date: September 25th, 6pm No late assignments will be accepted.
This assignment accounts for 8% of your final grade and is worth a total of 80 marks

The goal of this assignment is to practice using recursion and linked structures. You will decipher a piece of text that has been encrypted, and encipher the result using a different encryption method.

We use cryptography in the hope of being able to send a message secretly so that only the intended recipient can read the message. The original message is called *plaintext*. The encrypted message is called *ciphertext*. Traditionally plaintext contains only alphabetic characters, i.e. all white space characters are first removed. When using a *transposition cipher*, we encrypt by taking each consecutive piece of *m* characters of the plaintext and permuting them to produce *m* characters of ciphertext. Decryption is performed by applying the inverse permutation to each consecutive piece of the ciphertext.

Example: Permute each piece of 5 characters using permutation [31524]: the 1st character of ciphertext is the 3rd character of plaintext, the 2nd character of ciphertext is the 1st character of plaintext, etc.

Plaintext:	`meetatnoon`
Ciphertext:	`emaetotnno`

A few years ago, you encrypted some information using a transposition cipher and stored the resulting ciphertext in a file. You are unable to remember the permutation that you used but fortunately you do remember the first word in the plaintext and also that this first word has the same length as the permutation used (*m*). Even better, you are now a 2nd year Computer Science student who can write a program to decrypt the ciphertext. Since this is very secret information, you don't want to keep the plaintext as is, but rather encrypt it using a new method which will be explained later.

Your program must accomplish all of the following:

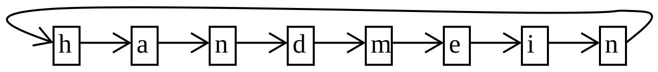
1. Read the ciphertext (*n* characters) and the known first word of plaintext (*m* characters). You may assume that *m* exactly divides *n*, i.e. you do not need to worry about partial permutations.
2. For each permutation of length *m*, apply the permutation to the first *m* characters of the ciphertext and output the corresponding plaintext. To find all permutations, you *must use a recursive method* that uses the following idea: a permutation of the values {v_1,. …, v_j} is found by taking each of the values v_1, …, v_j in turn and appending to it (in turn) each of the permutations of the remaining *j*-1 values.

   Example: the permutations of {1,2,3} are 123, 132, 213, 231, 312, 321. The output from permuting `obb` should be:
   ```
   [123] obb
   [132] obb
   [213] bob
   [231] bbo
   [312] bob
   [321] bbo
   ```
   Hint: you may wish to use a method `permute(j, vals[], …)` in which `vals[]` contains the values to be permuted, and that makes recursive calls to `permute(j-1, v[], …)` where `v[]` contains a subset of `vals[]`.

   Important notes for this method:
   - When testing it, be careful not to use too large a value of *m*, as *m*! permutations will be generated.
   - The method must be recursive, or you will receive zero marks for the method.
   - You may *not* assume that only one permutation will result in the known word.
3. For each permutation of the first *m* characters that matches the known word, apply that permutation to the entire ciphertext and output the result. Each of these is a potential plaintext.
4. For the *k*'th of these potential plaintexts (for a given value of *k*), store it in a circular linked list in which each node consists of a single character and a pointer to the next node.

5. Re-encipher this plaintext using the following idea: starting from the head of the circular linked list, remove every *i*'th character from the list (for a given value of *i*) and output it as the next character of ciphertext, until the list is empty.

   Example:



   When *i*=3, the circular linked list above will generate the ciphertext `nehmandi`.

Input: Your program should read from a file; this file has the following format:

A positive integer value *n*, indicating that we have a piece of ciphertext of length *n*.
A positive integer value *m*, indicating that the known first word (as well as the permutation used) is of length *m*. You may assume that *m* exactly divides *n*.
A piece of ciphertext of *n* characters. You may assume that this consists only of lower-case alphabetic characters.
A known word consisting of *m* characters. You may assume that this consists only of lower-case alphabetic characters.
A positive integer value *k* indicating that we want to re-encrypt the *k*'th potential plaintext.
A positive integer value *i*, indicating the value *i* to be used in the new encryption method.

Output:

All permutations of the first *m* characters of the ciphertext in the format shown below
For each of the permutations that match the known word, the corresponding potential plaintext
The new ciphertext corresponding to the *k*'th potential plaintext.

Example: the following input:

```
12
3
obbalcallcei
bob
2
4
```

should produce the following output:

```
[123] obb
[132] obb
[213] bob
[231] bbo
[312] bob
[321] bbo
boblaclaleci
bobcallalice
caeaibclllob
```

For submission purposes, you must hand in the output obtained by running your program using the input file `assn1in.txt` found on the course Sakai site. Your program *will* be tested on other input; therefore you must make sure that your program is thoroughly tested.

Submission Requirements:
All submissions will be via Sakai. All of the following must be put in a single, zipped folder for submission:
1. All program files required to run your program. All code must be commented and properly documented.
2. A "printout" of your program, which has been printed into PDF format. Ensure that the first page of this PDF document contains a statement specifying whether you used IntelliJ, Eclipse or DrJava.
3. A printout (in PDF) of the output for your program from the supplied input, `assn1in.txt`.