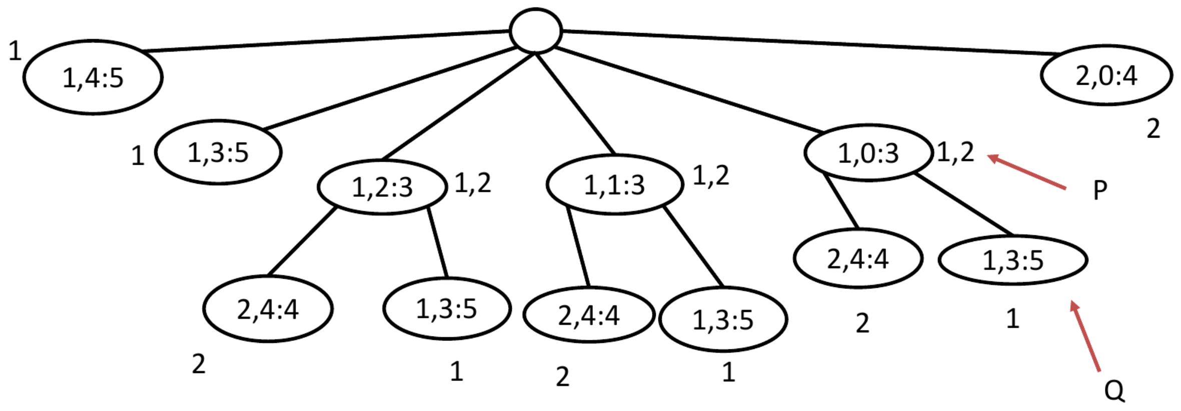
1) Consider the ADT **SuffixTree** with the following interface:

- ***SuffixTree(S)***: it creates a Suffix Tree starting from the tuple **S** of strings; each node of the tree, except the root, is *marked* with a reference to those strings in **S** for which there is a suffix going through node **u**; moreover, the substring associated to each node must not be explicitly represented in the tree (to this aim, **S** can be assumed to be an immutable object);
- ***T.getNodeLabel(P)***: it returns the substring that labels the node of **T** to which position **P** refers (it throws an exception if **P** is invalid);
- ***T.pathString(P)***: it returns the substring associated to the path in **T** from the root to the node to which position **P** refers (it throws an exception if **P** is invalid);
- ***T.getNodeDepth(P)***: it returns the length of substring associated to the path in **T** from the root to the node to which position **P** refers (it throws an exception if **P** is invalid);
- ***T.getNodeMark(P)***: it returns the *mark* of the node **u** of **T** to which position **P** refers (it throws an exception if **P** is invalid);
- ***T.child(P, s)***: it returns the position of the child **u** of the node of **T** to which position **P** refers such that
    - either **s** is a prefix of the substring labeling **u,**
    - or the substring labeling **u** is a prefix of **s**,

  if it exists, and it returns **None** otherwise (it throws an exception if **P** is invalid or **s** is empty).

For example, if **S = ("alive", "cali")**, then ***SuffixTree(S)*** builds the following tree **T**:



T.getNodeLabel(P) returns **"ali"**, while ***T.getNodeLabel(Q)*** returns **"ve"**.
T.pathString(P) returns **"ali"**, while ***T.pathString(Q)*** returns **"alive"**.
T.getNodeDepth(P) returns **3**, while ***T.getNodeDepth(P)*** returns **5**.
T.getNodeMark(P) returns **(1,2)**, while ***T.getNodeMark(Q)*** returns **1**.
T.child(P,"ve") returns **Q**, ***T.child(P,"vec")*** returns **Q**, ***T.child(P,"v")*** returns **Q**,
T.child(P,"o") returns **None**, ***T.child(P,"vo")*** returns **None**.

Provide an implementation of the ADT **SuffixTree** (and of the corresponding nested class **SuffixTree._Node**) such that:

- ***SuffixTree(S)*** has time complexity **O(n^2)** where **n** is the total length (i.e., the sum of the lengths) of strings in **S**. The returned suffix tree only contains **O(n)** nodes;

- ***T.getNodeLabel(P)***, ***T.getNodeMark(P)***, ***T.getNodeDepth(P)*** have time complexity **O(1)**;
- **T.pathString(P)** has time complexity **O(s)**, where **s** is the length of the substring associated to the path in **T** from the root to the node to which position **P** refers;
- ***T.child(P, s)*** has (amortized) time complexity **O(*len*(s))**.

2) Often the various laboratory processes used to isolate, purify, clone, copy, maintain, probe, or sequence a DNA string will cause unwanted DNA to become inserted into the string of interest or mixed together with a collection of strings. *Contamination* of protein in the laboratory can also be a serious problem. During cloning, contamination is often caused by a fragment (substring) of a vector (DNA string) used to incorporate the desired DNA in a host organism, or the contamination is from the DNA of the host itself (for example bacteria or yeast). Contamination can also come from very small amounts of undesired foreign DNA that gets physically mixed into the desired DNA and then amplified by PCR (the polymerase chain reaction) used to make copies of the desired DNA.

Contamination is an extremely serious problem, and there have been embarrassing occurrences of large-scale DNA sequencing efforts where the use of highly contaminated clone libraries resulted in a huge amount of wasted sequencing. These embarrassments might have been avoided if the sequences were examined early for signs of likely contaminants, before large-scale analysis was performed or results published.

Indeed, often, the DNA sequences from many of the possible contaminants are known. These include cloning vectors, PCR primers, the complete genomic sequence of the host organism (yeast, for example), and other DNA sources being worked with in the laboratory.

Consider then the following computational problem:

We are given a string **s** (the newly isolated and sequenced string of DNA) and a set **C** of known string $\mathbf{c_1}$, …, $\mathbf{c_m}$ (the possible contaminants). For each contaminant $\mathbf{c_i}$, the *degree of contamination* in **s** by **c**, is the number of maximal substrings of **c** that occur in **s** and that are longer than the given *contamination threshold* **l**, where a substring **x** of **c** is maximal if there is no substring **x'** of **c** that occurs in **s** and such that **x** is substring of **x'**.

For example, if **s="acgtatcgatg"**, **c="cgtgatga"**, and **l=3**, then the degree of contamination in **s** by **c** is **2** because **"cgt"** are **"gatg"** are common substrings of **s** and **c** of length at least **l**. Note that also **"gat"** and **"atg"** are common substrings of length at least **l**, but they must not be counted, since they are not maximal (they are contained in **"gatg"**).

You are required to identify the contaminants with higher degree of contamination in **s**. Specifically, you must implement a class **DNAContamination** that allows to verify the degree of contamination in a string **s** by a set of contaminants **C**. The class implements the following methods:

- ***DNAContamination(s, l)***: it build a **DNAContamination** object; it takes in input the string **s** to verify and the contamination threshold **l** (the contaminant set **C** is initially empty);
- ***addContaminant(c)***: it adds contaminant **c** to the set **C** and saves the degree of contamination of **s** by **c**;
- ***getContaminants(k)***: it returns the **k** contaminants with larger degree of contamination among the added contaminants.

The method ***addContaminant(c)*** should run in time **O((*len*(s)+*len*(c))^2 + log m)**, where **m** is the number of total contaminants that would be added. The method ***getContaminants(k)*** should run in time **O(k log m)**.

3) Implement a function ***test(s,k,l)*** that reads DNA strings from the dataset **target_batcha.fasta** and returns the indices of the **k** contaminants in the dataset with larger degree of contamination in **s**, assuming **l** as contamination threshold.

Specifically, the function must return a string containing these indices in increasing order separated by comma (each comma must be followed by space and no character must appear after the last index).

For example, on input **s = ”TGGTGTATGAGCTACCAGCCGTGCGAAACTCATACTATTATC TAATCAGGGACAATACCTCAGGCAGGACTGTGCTGTGTAGATAGCTGGAGAGTATTTC TGATTGTCTCCGAGGGGTGTAAAGGTACTTGCAAGGCCACTCAACTCATGCAGCGTTT CCATTTGAGTTGCCTTGAGTAAACGTCAACGCAGCTGGGAGTAGTACCTCTTGGAGGT TGTGACCGCCGCTGCCCGCATGGACAGACGCACGGAAATGTATTAACACTAACTATAC T”**, **k = 20**, and **l=7**, the function must return the string “**`1764, 2266, 5141, 9787, 10681, 10682, 12376, 12377, 12378, 12379, 12833, 12834, 12835, 12836, 14269, 14270, 18319, 18320, 18806, 19096`**”.[1]

The 15 projects that return the correct result in the shortest time will receive a bonus point.

---

[1] As sanity check, I will report for each contaminant **c** the starting index and the length of the substrings of **c** appearing in **s**:
1764: [(4, 7), (18, 7), (25, 7), (41, 7), (55, 7), (71, 7), (78, 7), (94, 7), (115, 7), (127, 7), (148, 7), (187, 7), (193, 7), (201, 7), (224, 7), (231, 7), (241, 7), (170, 8)]
2266: [(56, 7), (68, 7), (72, 7), (76, 7), (80, 7), (84, 7), (88, 7), (92, 7), (96, 7), (100, 7), (115, 7), (119, 7), (183, 7), (218, 7), (226, 7), (231, 7), (136, 8)]
5141: [(12, 7), (19, 7), (35, 7), (65, 7), (85, 7), (101, 7), (117, 7), (124, 7), (140, 7), (177, 7), (191, 7), (198, 7), (228, 7), (244, 7), (251, 7), (160, 8)]
9787: [(0, 7), (12, 7), (33, 7), (72, 7), (78, 7), (86, 7), (109, 7), (116, 7), (126, 7), (143, 7), (150, 7), (166, 7), (196, 7), (203, 7), (219, 7), (249, 7), (55, 8)]
10681: [(0, 7), (9, 7), (51, 7), (63, 7), (72, 7), (126, 7), (135, 7), (168, 7), (178, 7), (189, 7), (198, 7), (21, 8), (30, 8), (42, 8), (84, 8), (105, 8), (147, 8), (156, 9)]
10682: [(25, 7), (37, 7), (46, 7), (58, 7), (79, 7), (88, 7), (100, 7), (109, 7), (130, 7), (151, 7), (163, 7), (191, 7), (212, 7), (4, 8), (16, 8), (67, 8), (121, 8), (142, 8), (182, 8), (203, 8), (169, 10)]
12376: [(34, 7), (46, 7), (55, 7), (67, 7), (88, 7), (97, 7), (109, 7), (118, 7), (139, 7), (160, 7), (172, 7), (200, 7), (221, 7), (4, 8), (13, 8), (25, 8), (76, 8), (130, 8), (151, 8), (191, 8), (212, 8), (178, 10)]
12377: [(25, 7), (34, 7), (46, 7), (55, 7), (88, 7), (97, 7), (109, 7), (118, 7), (151, 7), (160, 7), (172, 7), (181, 7), (213, 7), (13, 8), (76, 8), (130, 8), (139, 8), (202, 9)]
12378: [(26, 7), (38, 7), (47, 7), (101, 7), (110, 7), (143, 7), (153, 7), (164, 7), (173, 7), (185, 7), (194, 7), (227, 7), (236, 7), (248, 7), (257, 7), (5, 8), (17, 8), (59, 8), (80, 8), (122, 8), (206, 8), (215, 8), (131, 9)]
12379: [(17, 7), (24, 7), (38, 7), (47, 7), (59, 7), (68, 7), (110, 7), (122, 7), (131, 7), (185, 7), (194, 7), (227, 7), (25, 8), (80, 8), (89, 8), (101, 8), (143, 8), (164, 8), (206, 8), (5, 9), (215, 9)]
12833: [(3, 7), (7, 7), (40, 7), (44, 7), (72, 7), (76, 7), (80, 7), (84, 7), (109, 7), (113, 7), (117, 7), (121, 7), (125, 7), (158, 7), (186, 7), (190, 7), (194, 7), (212, 7), (216, 7)]
12834: [(3, 7), (7, 7), (40, 7), (44, 7), (72, 7), (76, 7), (80, 7), (84, 7), (88, 7), (110, 7), (114, 7), (118, 7), (136, 7), (140, 7), (144, 7), (177, 7), (181, 7), (209, 7), (213, 7), (217, 7), (221, 7)]
12835: [(26, 7), (30, 7), (34, 7), (52, 7), (56, 7), (90, 7), (94, 7), (122, 7), (126, 7), (130, 7), (134, 7), (138, 7), (160, 7), (164, 7), (168, 7), (172, 7), (205, 7), (209, 7), (237, 7), (248, 7)]
12836: [(8, 7), (12, 7), (16, 7), (20, 7), (45, 7), (49, 7), (53, 7), (57, 7), (61, 7), (94, 7), (122, 7), (126, 7), (130, 7), (148, 7), (152, 7), (186, 7), (190, 7), (218, 7), (222, 7), (226, 7), (230, 7), (234, 7), (256, 7)]
14269: [(24, 7), (27, 7), (56, 7), (59, 7), (88, 7), (91, 7), (120, 7), (123, 7), (152, 7), (155, 7), (184, 7), (187, 7), (216, 7), (219, 7), (248, 7), (251, 7)]
14270: [(24, 7), (27, 7), (56, 7), (59, 7), (88, 7), (91, 7), (120, 7), (123, 7), (152, 7), (155, 7), (184, 7), (187, 7), (216, 7), (219, 7), (248, 7), (251, 7)]
18319: [(19, 7), (21, 7), (41, 7), (43, 7), (63, 7), (65, 7), (85, 7), (87, 7), (107, 7), (109, 7), (129, 7), (131, 7), (151, 7), (153, 7), (217, 7), (219, 7), (174, 8), (196, 8)]
18320: [(4, 7), (6, 7), (26, 7), (28, 7), (48, 7), (50, 7), (70, 7), (72, 7), (92, 7), (94, 7), (114, 7), (116, 7), (136, 7), (138, 7), (202, 7), (204, 7), (233, 7), (159, 8), (181, 8)]
18806: [(11, 7), (63, 7), (122, 7), (126, 7), (135, 7), (139, 7), (143, 7), (147, 7), (151, 7), (155, 7), (159, 7), (163, 7), (167, 7), (186, 7), (195, 7), (219, 7), (84, 9), (21, 10)]
19096: [(10, 7), (106, 7), (117, 7), (129, 7), (133, 7), (137, 7), (148, 7), (152, 7), (156, 7), (160, 7), (164, 7), (168, 7), (172, 7), (176, 7), (180, 7), (184, 7), (214, 9)]