**Анализ**

Първата подзадача е за 11 точки. Тук е достатъчно пълно изчерпване – правим два вложени цикъла за обхождане на всички поднизове и проверяваме за всеки дали няма друг подниз, разположен вляво, който е същия и ако има, то този вече е броен към отговора и не го броим втори път. Сложността е , но реално е много по-малко, защото няма как да има различни поднизове за по-голяма дължина, а и проверката за сравняване на поднизове ще е повечето пъти константна.

Втората подзадача е за 13 точки. Сега оптимизираме малко тази идея. Можем да приложим хеширане – да намерим хеш стойността на всеки подниз и да сложим всички стойности в един *unordered\_set*. Така големината му, ще е и броя различни поднизове. Сложността е .

Третата подзадача е за 10 точки. Тук вече можем да имаме заявки. Пак използваме горното решение, като при всяка заявка актуализираме *unordered\_set* с хешовете. Сложността е .

Четвъртата подзадача е за 12 точки. Това е сравнително известна задача, за която трябва вече да използваме суфиксна структура. Нека за всяка заявка намираме суфиксния масив на текущия низ. Тогава можем да намерим отговора по следния начин. Обхождаме суфиксите в подредения вид. Първо добавяме броя букви на първия суфикс (така преброяваме всички поднизове, които са му префикси). След това като сме на втория суфикс и пак добавяме дължината му. Но така евентуално сме преброили някои поднизове по няколко пъти. Това са общите префикси на първия и втория подниз. За да намерим повторенията използваме *lcp* (longest common prefix) функцията за тези два суфикса. От отговора вадим полученото от функцията и продължаваме по същия начин да обхождаме подредените суфикси. Ясно е, че по този начин, понеже суфиксите са подредени ще преброим всички различни поднизове, защото в тази наредба съседите на даден суфикс са най-близките в лексикографско отношение суфикси. Тази подзадача не дава повече точки, защото това е сравнително стандартна задача. Сложността е – имаме , защото ако всички операции са по прибавяне, то дължината на низа само нараства.

Петата подзадача е за 15 точки. Тук ще направим важна крачка към пълното решение. Ще оптимизираме горната идея (ясно е, че в предната подзадача имаме най-тривиалния начин за решение, защото генерираме всеки път суфиксния масив на низа). Нека помислим как се променя суфиксния масив при заявките. Ако махаме отпред, то просто махаме един суфикс (целия досегашен низ) и като цяло суфиксите остават по същия начин. Не така стои въпроса с добавянето отзад. Като добавим буква, тогава всички суфикси се увеличават с нея и съответно може да имаме размествания (при тези суфикси, които са префикси на съсед), които не са лесно предвидими. Затова ще направим стандартен трик, като се възползваме, че заявките са в офлайн режим. Ще ги гледаме в обърнат вид, т.е. първо ги прочитаме и извършваме операциите с низа и така получаваме крайния низ. След което, като ги гледаме наобратно имаме следните операции – операцията по добавяне става по премахване отзад, а операцията по премахване става по добавяне отпред. Отново е ясно, че нямаме проблем с операцията добавяне отпред. Въпросът е при другата какво да правим. Тя отново промяна всички суфикси, като маха една буква отзад. Отново може да имаме промяна в наредбата за суфикси, които са префикс на съсед. Но тук има нещо много важно – при тези операции суфиксите само намаляват като дължина. А ако имаме суфикс, който е префикс на съсед, той с нищо не допринася за отговора по досегашния алгоритъм. Това означава, че можем просто да премахваме всеки суфикс, който стане префикс на съсед. Така няма да имаме промяна в наредбата при операция по премахване! Можем да направим следния алгоритъм – в един булев масив пазим в подредения вид на суфиксите кои все още ни вършат работа. Като имаме операция по добавяне, слагаме единица в този масив на съответната позиция и освен това обхождаме всички суфикси, за да видим дали евентуално няма суфикс, който е станал префикс на съсед при добавянето. При операцията по премахване, слагаме нула в булевия масив на съответната позиция и пак обхождаме масива, за да видим дали не отпадат нови суфикси. Също като гледаме дали някой низ не е префикс на съсед сравняваме дължините и отново ползваме *lcp* функцията. Понеже искаме да оптимизираме *log* множителя, трябва да използваме, че *lcp* на два суфикса представлява *RMQ* заявка в lcp масива за даден низ. Така можем да направим със *sparse table* *RMQ*-то с константа заявка. Тук сложността е .

Шестата подзадача е за 13 точки. При нея имаме само операции по премахване. Отново използваме трика да гледаме заявките в обратен ред. Така тези операции стават само по добавяне. Както отбелязахме при тях просто се добавя нов суфикс в суфиксния масив. Понеже няма промяна в наредбата освен добавяне на нов суфикс, то няма нужда да следим дали някой суфикс става префикс на съсед. Така можем просто да използваме *set*, за да пазим номерата на суфиксите, които са за текущия низ. След което, като сложим нов суфикс гледаме двата съседа (или само един, ако е в някой край) и променяме отговора подобаващо. Така сложността е .

Седмата подзадача е за 15 точки. Тук трябва да се постараем да реализираме по-добре идеята, която говорихме в четвърта подзадача. Там просто обхождахме всеки път текущите суфикси. Отново ще използваме *set*, за да пазим номерата (разбира се номерата в сортираната наредба) на суфиксите, които са за текущия низ. Някак трябва да следим постоянно дали някой суфикс не става префикс на съсед. Затова ще използваме един *multiset*, в който ще пазим наредена двойка, която е отношението между суфикс и някой съсед. Първото число, ще е броя символи на суфикса без общата част със съседа му, а второто ще е номера на суфикса (за да можем да следим и кой суфикс отпада, все пак). Така ако първото число на някоя наредена двойка стане 0, означава, че даден подниз е станал префикс на съсед и трябва да го махнем. Второто число ще е позицията на суфикса в суфиксния масив, това ни трябва задължително, за да може като трябва да махаме повече от един суфикс да става подредено, а не хаотично, което би създало проблеми. Можем да направим функция, която гледа дали първият елемент на *multiset*–а няма първо число 0 и ако има, то някой суфикс отпада и така върти цикъл, докато не се появи по-голямо число или не свършат суфиксите (при премахване на всички букви например). Отпадането на суфикс означава да преизчислим отговора, като видим кои са му съседите. Сега нека разгледаме операциите. Като имаме операция по добавяне трябва да добавим съответния суфикс в *set*-а на суфиксите и после да извикаме функцията, която проверява за суфикси, които са префикси на съсед. А като имаме операция по премахване трябва да намалим всички числа в *multiset*–а с единица и да видим какво се случва. Тук може да изглежда, че ще извършваме многократни махания и това е така. Но ако разгледаме амортизирано, то всеки суфикс, го гледаме два пъти – при добавяне и премахване, т.е. амортизирано е константа операцията (по лог, заради работата със *set*-а) Ясно, е че това нямам как да стане лесно, затова просто ще имаме една променлива, която следи броя премахнати букви. За да работи това, в *multiset*–а ще слагаме дължината на *lcp*-то, като гледаме суфиксите все едно са продължени до края на низа (т.е. все едно не сме махали букви). Така описания алгоритъм е със сложност , но с доста голяма константа за всяка операция.

Последната подзадача е за 11 точки. Тук само ще подобрим константата. Вместо *set*-а за суфиксите ще използваме дърво на Фенуик (това е все едно надграждане на идеята с булевия масив). За да можем да намираме съседи на даден суфикс в дървото на Фенуик, трябва да можем за дадена единица да намерим най-близката вляво и вдясно. Това можем да направим с лог^2 стъпки с лесно двоично търсене, но тук ще използваме по-добра идея – подобие на *binary lifting*. Нека искаме да намерим най-малкия префикс със сума *s* (което е все едно на позиция *s* кой е суфикса). Почваме от началото на масива и от най-голямата степен, която е ≤ ***N***+***Q***. Проверяваме дали като я добавим няма да получим по-голяма сума от *s*, ако е така пробваме с по-малка степен и повтаряме тази стъпка. Това гледане е просто да проверим числото в масива, който е дървото на Фенуик с позиция, отместена с колкото е степента на двойката (това е така, заради специфичния начин, по който всяка позиция отговаря за даден интервал). Разбира се, тук ще ползваме *RMQ* с константа заявка за намиране на *lcp* на два суфикса, което ще ни трябва неколкратно. Другото, което ще оптимизараме е *multiset*-а. Отново с дърво на Фенуик (все пак при него имаме чист логаритъм за всяка операция). Първото число от наредената двойка сега ще е индекса в масива, а всеки елемент ще е *multiset*, но много по-малък от досегашния. Не става всеки елемент да е *vector* например, защото трябва да махаме подредено суфиксите. Отново ще използваме трика за *lower\_bound* във дървото на Фенуик, като тук ще ни трябва само, за да намерим сума 1 и да видим дали индекса не е по-малък от премахнатите букви. С тези оптимизации, които споменахме значително намаляваме константа и разбира се отново сложността е .

*Автор: Илиян Йорданов*