Проблем 1. О-речник

Мали Декица је одлучио да направи свој први речник. У њега ће да смешта све речи којих се докопа. Наравно као што је у сваком речнику правило, речи су послагане по лексикографском поретку од најмање до највеће. Временом речник расте, па би мали Декица волео да зна за неку реч, колико мањих речи од ње има у речнику.

Улаз. (Улазни подаци се учитавају на стандардног улаза) У првоме реду улаза налази се један цео број N ($1 \leq N \leq 100.000$) који представља број команди. У следећих N редова, налазе се нека од команди

ADD rec

LESS rec

Речи ће бити састављене од слова енглеске абецеде, било малих било великих, при чему се сматра да су речи "*PoPoKaTaPeTL*" и "*POPOkataPETl*" исте. Ниједна реч неће бити дужа од 100 знакова, а комплетан речник неће имати више од 2.000.000 слова.

Излаз. (Излазне подаци се исписују на стандардни излаз) На стандардни излаз треба за сваку команду која почиње са *LESS* исписати по један цео број који представља број речи лексикографски мањих од речи која следи иза команде *LESS*. Сваки број штампати у новом реду. Ако тражене речи нема у речнику исписати *"no such word"*.

Пример 1.

proizvod.in	proizvod.out
14	2
ADD Petronije	no such word
ADD PeTrONIJE	0
ADD Jovan	6
ADD STEVAN	
LESS Stevan	
ADD ISTVAn	
LESS pajVan	
ADD maja	
LESS istvan	
ADD KlipaN	
ADD Milica	
LESS stevan	
ADD Milica	
ADD MiliCA	

Напомена. У 50% примера N ће бити мање или једнако 1000.

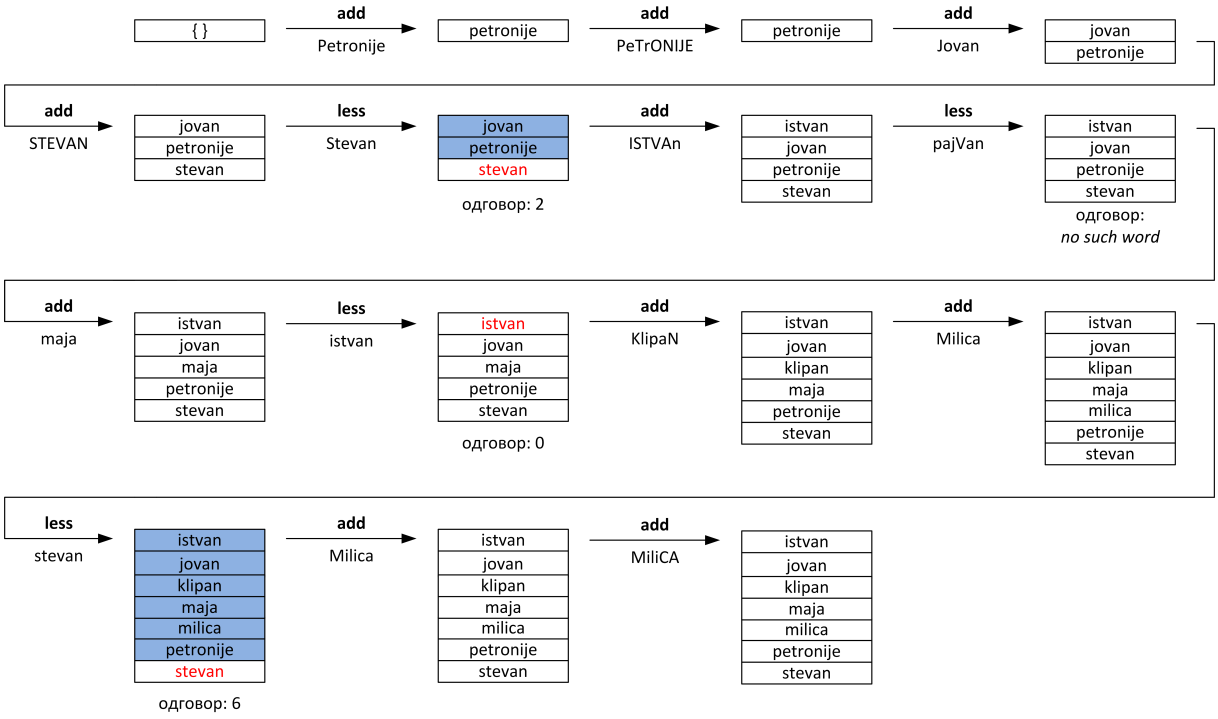
Ограничења. Временско ограничење: $1s$; Меморијско ограничење: $64MB$

Решење и анализа. Први проблем са ових квалификација, иронично, представља и најтежи проблем. Наиме, он једини захтева познавање теорије односно како ћемо видете неких напреднијих стуктуру података. Просечан број бодова је био око 15 при чему је једанаест такмичара имало максималних 100 бодова на њему.

Проблем је доста нејасан након првог читања (што потврђује и доста постављених питања на форуму у току самог такмичења). Међутим, скоро све сумње отклања пример дат у самом проблему (што и јесте сврха примера са папира). У грубим цртама, од нас се захтева да имплементирамо програм који симулира следећи алгоритам:

1. Постави речник, означићемо га са *recnik*, на празан скуп.
2. За сваку од n команди, у редоследу датом као у улазу:
   (а) Команда облика *ADD rec*: *rec* трансформисати тако да садржи само мала слова енглеске абецеде (пошто речи нису *case sensitive*). Уколико *rec* не припада *recnik*-у, убацити је.
   (б) Команда облика *LESS rec*: *rec* трансформисати тако да садржи само мала слова енглеске абецеде (пошто речи нису *case sensitive*). Уколико *rec* не припада *recnik*-у, вратити -1 као резултат; у супротном вратити број речи из речника које су мање од ње.

Под "мање" подразумевамо мање у односу на лексикографски поредак. У даљем делу текста, подразумеваћемо да су речи из улаза трансформисане тако да се састоје само од малих слова. Симулација овог алгоритма за дати пример је приказана на слици 1. На њој је *recnik* у сваком од корака приказан у растућем поретку (ускоро ћемо видети зашто нам је поредак битан).



Слика 1. Објашњење примера са папира.

Овде ћемо изложити два решења овог проблема. Прво, једноставније решење, које је и већина такмичара имплементирала, ћемо детаљно образложити, док ће друго бити изложено у кратким цртама (више због саме идеје). Битно је истаћи да је метод који се користи у првом решењу јако битан и надасве користан.

Као што је и у горњем алгоритму описано, потребно је у сваком кораку оджавати структуру *recnik* (још увек не знамо како она треба да изгледа) тако да можемо: (*i*) додати нову реч; (*iI*) испитати да ли одређена реч постоји у њој; (*iii*) наћи број речи који су мањи од дате речи из речника.

Код проблема овог типа, где се одржава структура и потребно је имплементирати одређен број операција и упита, могућа су два генерална приступа:

- **on-line** приступ: Овде се команде учитавају редом. Уколико је команда заправо упит, одмах се штампа њен одговор.
- **off-line** приступ: Овде се прво све команде учитају и креира се структура на основу њих (другим речима "изглед" структуре зависи од самог улаза). Затим се поново креће по командама и тек тада штампа резултат за одређени упит.

Питање које се овде намеће јесте: Зашто би нама користио *off-line* приступ? Одговор нећемо дати у генералном случају (изгледао би сувише филозофски овако у почетку, а то свакако не желимо), већ ћемо то урадити кроз решење овог проблема.

Off-line решење: Претпоставимо да смо на почетку учитали све команде. Команде можемо запамтити у облику сруктуре *command* која има следеће атрибуте:

атрибут	објашњење
isAdd	*boolean* који служи да разликујемо команде (узима вредност *true* уколико је команда за додавање речи, иначе је $false$)
word	реч коју треба убацити или за коју треба извршити упит
index	редни број односно позиција команде у улазу
sort	индекс у сортираном речнику (у даљем делу објашњавамо овај атрибут)

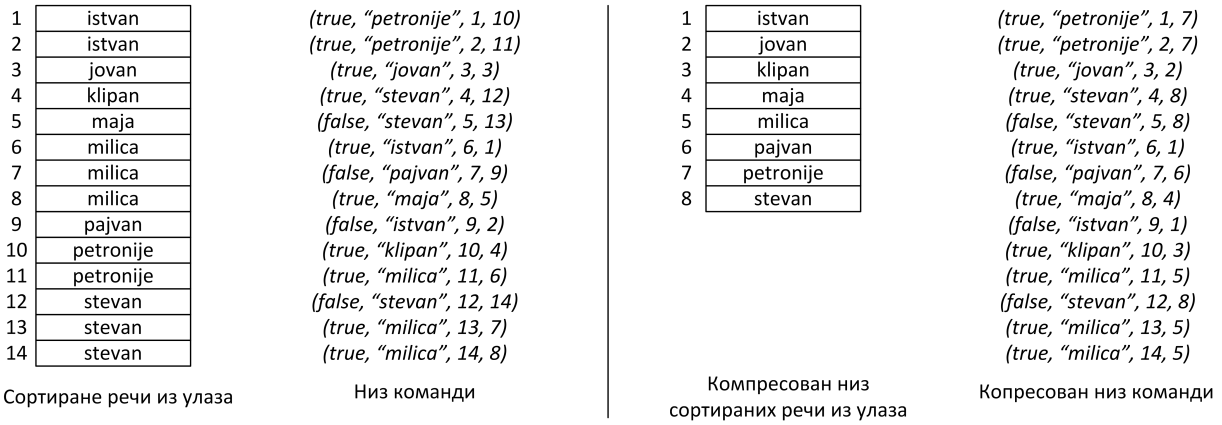
Дакле, при само учитавању креирамо низ команди и за сваку од њих иницијализујемо прва три атрибута. Овај низ команди, можемо сортирати лексикографски по атрибуту *word* односно по речима на које се команде односе. Шта добијамо овим? Овим се добија управо поредак речи на крају односно након извршења свих команди (при чему се и речи из упита налазе у њима). Зашто нам је потребан овај поредак? Наиме, како се упит односи на број речи које су мање од дате, ми на неки имплицитни начин морамо држати речи из речника сортиране (пошто је број речи које су мање заправо индекс упитне речи у сортираном низу умањен за један).

Ако би при сваком додавању нове речи у структуру, поново сортирали (односно пошто је она већ сортирана убацивање извршавамо уметањем у линеарном времену), добили би да је сложеност убацивања нове речи пропорционалан броју речи у речнику. Овде кажемо пропорционалан из простог разлога што овде баратамо са стринговима па операција упоређивања није константне сложености. Заиста, за упоређивање два страинга, у најгорем случају, потребан број операција једнак је дужини краће речи.

Уколико би ми имали крањи изглед речника, ми можемо реч убацити на место на којем се она налази на крају (тачније увек одржавамо поредак речи). На овај начин добијамо да у сваком тренутку имамо сортиран низ речи али имплицитно, јер између речи може бити

празних места (места која ће тек бити попуњена речима које касније долазе). Зато након уноса команди, сортирамо низ по атрибуту *word*.

Чему нам служи атрибут *sort*? Наиме, нама није битно да имамо низ команди сортиран по атрибуту *word*. Као што ћемо ускоро видети, потребно је само да за сваку од команди знамо њен индекс у сортираном низу. Дакле, у атрибут *sort* памтимо управо овај индекс.



Слика 2. Изглед сортираног низа речи и низа команди (у почетном и копресованом облику).

Међутим, на овај начин остављамо више поља за исту реч (пример ради у улазу се налази три пута реч *stevan*). Зато након овго основног сотирања вршимо "копресију" овог низа. Под компресијом мислимо да сва понављања речи избацујемо. Како је низ речи сортиран, ово можемо урадити у линеаном времену односно једним проласком кроз низ (Како?). Након овога мењамо и вредности атрибута *sort* у нашим командама.

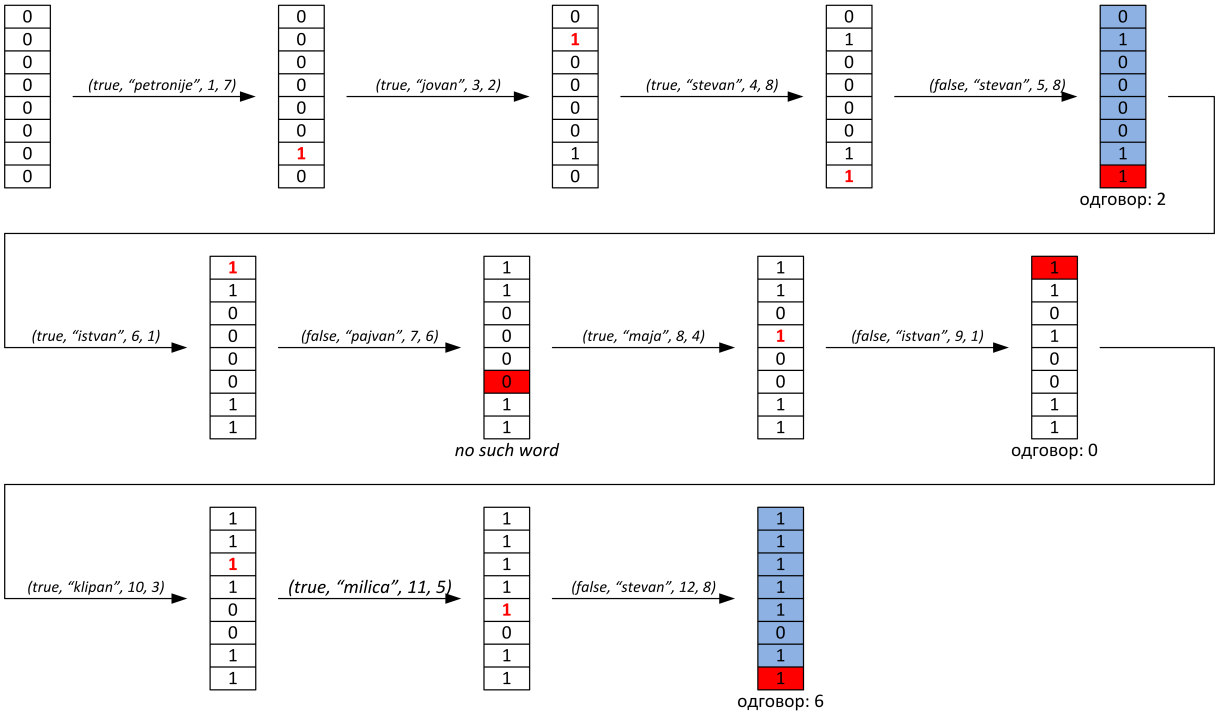
Проблем смо свели на симулацију следећег алгоритама:

1. Иницијализовати низ команди, *commands*. Атрибут *sort* на почетку постављамо да има исту вредност као и *index*.
2. Сортирати низ команди, индиректно преко индекса *sort* (дакле не мењамо редослед команди команде већ вредности овог атрибута).
3. Компресија низа *commands* односно вредности атрибута *sort*.
4. Поставимо низ стрингова *words*, дужине копресованог низа речи, на празне речи. Ово ће нам представљати стање структуре односно служиће нам као тренутни сортирани низ речи са "рупама".
5. Поново се крећемо по низу команди (по редоследу са улаза). Када се налазимо на k-тој команди:
   (а) ако је команда типа *add* односно ако је $commands[k].add = true$, тада постављамо $words[commands[k].sort] = commands[k].word$. Овим реч одмах постављамо управо на позицију коју ће имати на крају.
   (б) ако је команда типа *less* односно ако је $commands[k].add = false$, тада прво треба испитати да ли је реч убачена у речник. Ово испитујемо преко услова да је $word[commands[k].sort]$ различито од празног стринга. Уколико није, као резултат враћамо *no such word*. У супротном потребно је вратити број речи пре ње, а ово је управо број не празних стрингова у низу *words* пре индекса $commands[k].sort$.

У суштини, низ *words* не мора бити низ стрингова. Њега смо тако дефинисали само ради лакше аналогије са сортираним низом речи. Сада када знамо шта он заправо представља, можемо га посматрати као низ целих бројева. На почетку све елементе иницијализујемо на нуле. Када се нека реч убацује на позицију k, ми заправо постављамо $words[k]$ на један. Овим се упит (број речи пре ње које су постављене на 1) своди на суму елемената низа пре неког индекса. Звучи познато? Овај проблем се решава **кумулативном табелом**.

Један од честих проблем које срећемо у програмирању јесте проблем кумулативних табела, који се заснива на следећим операцијама над динамичким низом:

- $add(k, x)$ - повећај k-ти члан низа за x
- $count(k)$ - наћи суму првих k елеманата низа



Слика 3. Симулација команди над кумулативним низом. Задње две команде (које не утичу на резултат а и не мењају структуру) нису приказане.

Очигледно "решење" је да у низу a чувамо вредности низа, а суму првих k чланова добијамо сумирањем првих k елемената тог низа. То са собом повлачи сложености $O(1)$ и $O(n)$ за обе операције, редом. Друга могућност јесте да се у додатном низу *sum* памте префиксне суме. Тада добијамо обрнуте сложености за операције (за свако додавање k-том елементу морамо да променимо и вредности свих сума индекса већег или једнаког k). До оптималнијег решења долазимо уколико покушамо да ове сложености уједначимо. Структура података која решава овај проблем у сложености $O(n \log n)$, за обе операције, јесу поменуте кумулативне табеле. Овде нећемо описати алгоритам кумулативних табела. Читаоцу предлажемо да се са истим упозна у другој литератури.

"Уопштење" кумулативних табела је сутруктура: **сегментно стабло** (енг. *segment tree*). Сегментним стаблом се такође могу рачунати префиксне суме у логаритмаском времену (између

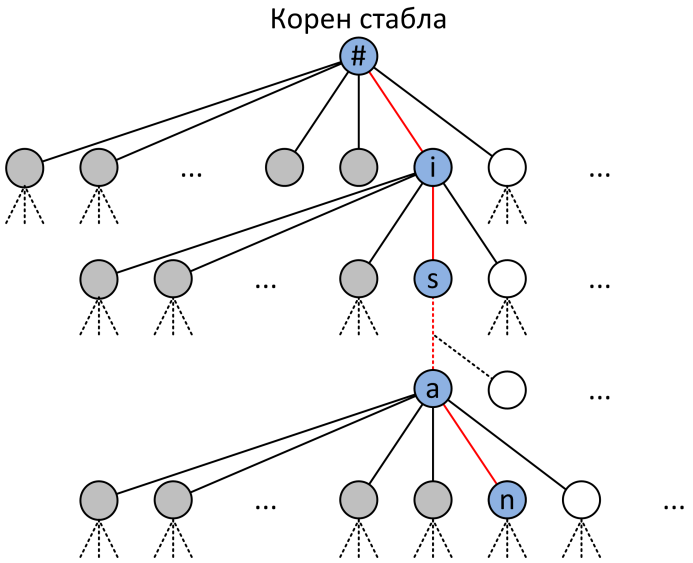
осталог), тако да се и њиховом имплементацијом добија иста сложеност. Наравно, препоручујемо да увек примењујете ону у чијем друштву се сигурније осећате.

Дакле, дефинисањем низа *word* преко кумулативне табеле, добијамо коначан алгоритам. Због мноштва корака које имамо у алгоритму, поставља се питање која је коначна сложеност овог алгоритма. Први корак, односно сортирање низа команди, има сложеност $O(n \log n \cdot 20)$. Одакле 20 у овој формули? Па већ смо напоменули да пошто се овде играмо са стринговима, сложеност упоређивања је једнак дужини мањег стринга. Ограничење за дужину стринга је 100 међутим просечна дужина стринга је 20 (напоменуто је да укупна дужина не прелази два милиона карактера). Други корак, који се односи на компресију има сложеност $O(n \cdot 20)$ (линеарна сложеност али параметар упоређивања је опет заступљен). На крају остаје део везан за симулацију, који се своди на директну примену кумулативне табеле. Коначно, добијамо да је сложеност нашег алгоритма једнака $O(n \log n \cdot 20 + n \cdot 20 + n \log n) = O(20 \cdot n \log n)$.

On-line решење: Као што смо напоменули на почетку, постоји још један начин за решавање овог проблема. Кључна разлика између ових приступа је што је ово *off-line* решење. Другим речима, у тексту проблема се могло захтевати да се одмах након уноса команде *less* штампа резултат. Тада, описани алгоритам не би био решење (пошто он захтева унос свих команди у старту). Међутим, обично је *off-line* приступ доста лакши.

Овде ћемо укратко изнети идеју алгоритма, али је нећемо детаљно описивати. Одмах у старту треба напоменути да је, за овај конкретан пример, јако тешко имеплентирати овај алгоритам зато што је меморијско ограничење овог проблема јако мало - $64MB$. Но, надамо се да ће вам ова идеја пробудити машту и да ће вам бити још једано оружје у арсеналу.

Као и прво решењу, и овде ћемо користити једну напреднију структуру података: **trie**. Детаљнији опис ове структуре је издвојен и дат у прилогу А. Све речи које се додају у речник, убациваћемо у наше стабло. Када наиђемо на упит *less* за реч *word*, прво је потребно испитати да ли се реч налази у речнику. Ово радимо једноставним кретањем од корена, а пратећи карактере речи *word*. Уколико се реч налази у стаблу, завршавамо у маркираном чвору, који ћемо означити са v.



Слика 4. Приказ стабла и рачунања одговора за команду *less*. Плавом бојом су обојени чворови по којима се крећемо, а сивом они за које сабирамо *count* вредности.

Како можемо окарактерисати речи које су мање од *word* која се налази у чвору v (под "налази" подразумевамо да се реч при кретању од корена завршава у том чвору)? Посматрајмо реч w којој одговара чвор u. Означимо са p првог заједничког родитеља за ова два чвора. Речи w ће бити лексикографски пре речи *word* ако и само ако се подстабло родитеља p коме припада u налази лево од подстабла коме припада v. Овде треба издвојити специјалне случајеве када је $p = v$ или $p = u$. Дакле, одговор за *less* добијамо тако што при кретању кроз стабло, по карактерима речи *word*, када прелазимо са чвора v на чвор u сабирамо *count*-ове све деце чвора v која су пре u (за додатно објашњење погледати слику 4).

Сложеност убацивања речи једнак је њеној дужини. Сложеност упита једнак је дужини речи помноженој са 26. Множимо са 26 због сумирања *count* атрибута "рођака" који су лево од чвор који посећујемо. Ако са *len* означимо укупну дужину улаза, односно укупну дужину речи, сложеност овог алгоритма је $O(len \cdot 26)$. Још једном напомињемо да је за овај конректан проблем јако тешко имплементирати ову идеју због малог меморијског ограничења, али се надамо да сте уживали у њој.