

Анализа проблема са решењима

Квалификације, као додатно такмичење у годишњем циклусу такмичења из програмирања ученика средњих школа, уведено је 2008. године. Два главна мотива за увођење додатног ранга такмичења су:

1. **уознавање са системом такмичења** (за почетнике)
2. **додатни тренинг ”старијим” такмичарима** (и свима који се тако осећају)

Настава из информатике, у нашим средњим школама, је организована на такав начин да је акценат далеко од проблема алгоритамске природе. Због овога, често ученици имају проблема са разумевањем концепта задатака који су заступљени на такмичењима из програмирања. На почетку сваког циклуса постоје питања око читавања / исписивања података (било са стандардног улаза / излаза било из датотека), такмичари нису упознати са системом тестирања, бодовањем проблема итд. Због тога се контролни задаци и одржавају тако да користите аутоматски евалуатор Вашег кода.

Погледајте пар приручника који нуде савете како да избегнете уобичајене грешке на такмичењима:
<http://poincare.matf.bg.ac.rs/~jelenagr/1d/OkruznoTakmicenjeNapomene.pdf>

У овом документу су дата решења у псеудо-језику и анализе проблема са Квалификација.

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 првome 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 абецедe, 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.

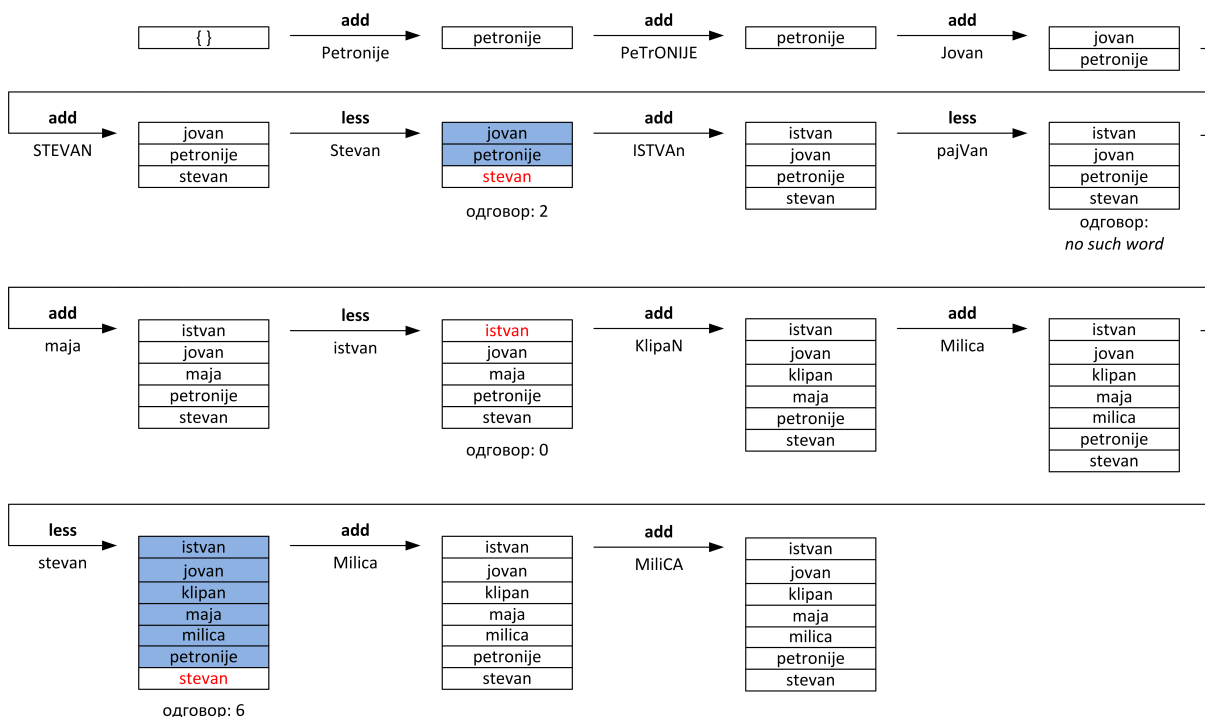
Ограничења. Временско ограничење: 1s; Меморијско ограничење: 64MB

Решење и анализа. Ово је пример тежег проблема на Квалификацијама, јер захтева да имате предзнања из неких напреднијих структура података. Али, не брините се. Имате 24 часа да сами мало истраите технике којима се напада проблем као што иначе раде прави информатичари. 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 који се дају уз формулацију задатака). У грубим цртама, од нас се захтева да имплементирамо програм који симулира следећи алгоритам:

1. Постави речник, означимо 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 *resnik* (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 читали све команде. Команде можемо запамтити у облику структуре *command* која има следеће атрибуте:

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

Дакле, при само читавању 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 добијемо 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 краћи изглед речника, ми можемо реч убацити на место на којем се она налази на крају (тачније увек одржавамо поредак речи). На овај начин добијемо 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.

<table border="1"> <tr><td>1</td><td>istvan</td></tr> <tr><td>2</td><td>istvan</td></tr> <tr><td>3</td><td>jovan</td></tr> <tr><td>4</td><td>klipan</td></tr> <tr><td>5</td><td>maja</td></tr> <tr><td>6</td><td>milica</td></tr> <tr><td>7</td><td>milica</td></tr> <tr><td>8</td><td>milica</td></tr> <tr><td>9</td><td>pajvan</td></tr> <tr><td>10</td><td>petronije</td></tr> <tr><td>11</td><td>petronije</td></tr> <tr><td>12</td><td>stevan</td></tr> <tr><td>13</td><td>stevan</td></tr> <tr><td>14</td><td>stevan</td></tr> </table>	1	istvan	2	istvan	3	jovan	4	klipan	5	maja	6	milica	7	milica	8	milica	9	pajvan	10	petronije	11	petronije	12	stevan	13	stevan	14	stevan	<p>(true, "petronije", 1, 10) (true, "petronije", 2, 11) (true, "jovan", 3, 3) (true, "stevan", 4, 12) (false, "stevan", 5, 13) (true, "istvan", 6, 1) (false, "pajvan", 7, 9) (true, "maja", 8, 5) (false, "istvan", 9, 2) (true, "klipan", 10, 4) (true, "milica", 11, 6) (false, "stevan", 12, 14) (true, "milica", 13, 7) (true, "milica", 14, 8)</p>	<table border="1"> <tr><td>1</td><td>istvan</td></tr> <tr><td>2</td><td>jovan</td></tr> <tr><td>3</td><td>klipan</td></tr> <tr><td>4</td><td>maja</td></tr> <tr><td>5</td><td>milica</td></tr> <tr><td>6</td><td>pajvan</td></tr> <tr><td>7</td><td>petronije</td></tr> <tr><td>8</td><td>stevan</td></tr> </table>	1	istvan	2	jovan	3	klipan	4	maja	5	milica	6	pajvan	7	petronije	8	stevan	<p>(true, "petronije", 1, 7) (true, "petronije", 2, 7) (true, "jovan", 3, 2) (true, "stevan", 4, 8) (false, "stevan", 5, 8) (true, "istvan", 6, 1) (false, "pajvan", 7, 6) (true, "maja", 8, 4) (false, "istvan", 9, 1) (true, "klipan", 10, 3) (true, "milica", 11, 5) (false, "stevan", 12, 8) (true, "milica", 13, 5) (true, "milica", 14, 5)</p>
1	istvan																																														
2	istvan																																														
3	jovan																																														
4	klipan																																														
5	maja																																														
6	milica																																														
7	milica																																														
8	milica																																														
9	pajvan																																														
10	petronije																																														
11	petronije																																														
12	stevan																																														
13	stevan																																														
14	stevan																																														
1	istvan																																														
2	jovan																																														
3	klipan																																														
4	maja																																														
5	milica																																														
6	pajvan																																														
7	petronije																																														
8	stevan																																														
Сортиране речи из улаза	Низ команди	Компресован низ сортираних речи из улаза	Компресован низ команди																																												

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 овог основног сортирања вршимо "копресију" овог низа. Под копресијом 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.

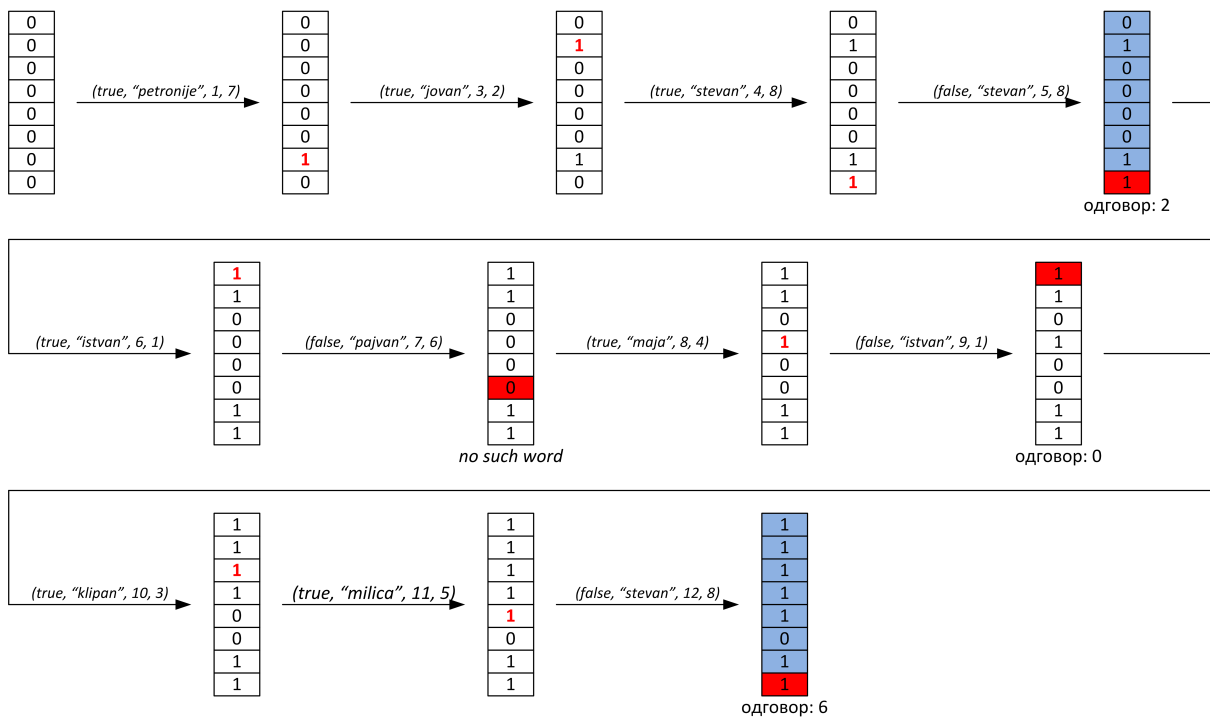
Проблем смо свели на симулацију следећег алгоритама:

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 **кумулятивном 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 елемената 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 структура: **сегментно стабло** (енг. *segment tree*). Сегментним стаблом се такође могу рачунати префиксне суме у логаритамском времену (између

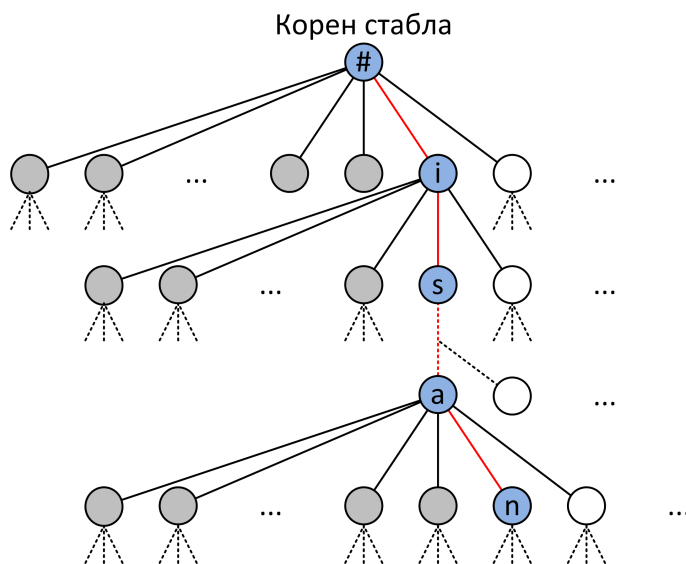
осталог), тако да се и њиховом имплементацијом добија иста сложеност. Наравно, препоручујемо да увек примењујете ону у чијем друштву се сигурније осећате.

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 имплементирати 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 алгорита је $O(len \cdot 26)$. Још једном напомињемо да је за овај конкретан проблем јако тешко имплементирати ову идеју због малог меморијског ограничења, али се надамо да сте уживали у њој.

Степен

За два дата природна броја a и b ($2 \leq a \leq b < 2^{64}$) одредити природне бројеве x и k тако да важи $a \leq x^k \leq b$, а да је k што веће могуће. У случају да има више решења исписати оно у којем је x најмање.

Улаз. (Улазни подаци се учитавају на стандардног улаза) У првој и јединој линији стандардног улаза се налазе природни бројеви a и b .

Изаз. (Изазне подаци се исписују на стандардни излаз) На стандардни излаз исписати редом два тражена природна броја x и k .

Пример 1.

stepen.in	stepen.out
5 20	2 4

Пример 1.

stepen.in	stepen.out
35 50	6 2

Ограничења. Временско ограничење: 0.5s; Меморијско ограничење: 16B

Решење и анализа. Други проблем на квалификацијама био је математичке природе. Решење проблема се врло једноставно може исказати у виду математичког алгорита. Међутим како се ми овде играмо у бинаром и ограниченом свету, није све тако лако као што изгледа. Просечан број бодова у току такмичења био је око 28, док је 50% такмичара освојило позитиван број бодова на њему. Само шеснаест такмичара је имало максимални учинак.

Прва ствар коју треба приметити јесте да су могуће вредности степена k дискретне.

Наиме, како су границе сегмента ограничене са 2^{64} , а имајући у обзир чињеницу да је x природан број, имамо да тражени степен k не може бити већи од 63. Дакле, k узима вредности из скупа $\{1, 2, \dots, 63\}$.

У тексту проблема се захтева минимизација број x за тражено k . Препоставимо да смо степен k фиксирали. Које су могуће вредности за број x ? Уколико дати услов $a \leq x^k \leq b$ мало трансформишемо, добијамо да важи

$$\sqrt[k]{a} \leq x \leq \sqrt[k]{b}$$

Како је x природан број, минимална вредност која задовољава горње неједнакости може имати два облика:

$$\lfloor \sqrt[k]{a} \rfloor \quad \text{или} \quad \lfloor \sqrt[k]{a} \rfloor + 1$$

Једноставним испитивањем да ли прва односно друга могућност задовољава и горњу границу, налазимо тражено x за дато k . Како k може имати само 63 могућих вредности, испитивањем сваке могућности долазимо до коначног решења. Овде треба напоменути да случај $k = 1$ треба посебно да се издвоји. Пошто се тражи максимално k , можемо кренути од максималне могуће вредности, односно 63, и редом испитивати за вредност x . Чим наиђемо на прво k за које постоји x које задовољава услове, програм може прекинути рад и вратити ове вредности.

Алгоритам: Псеудо код алгоритма проблема Степен

Input: крајеви сегмента a и b

Output: тражене вредности x и k

```
for  $k \leftarrow 63$  to 2 do
     $x = \lfloor \sqrt[k]{a} \rfloor$ ;
    if ( $x^k < a$ ) then
        |  $x = x + 1$ ;
    end
    if ( $a \leq x^k \leq b$ ) then
        | return ( $x, k$ )
    end
end
return ( $a, 1$ )
```

Дата ограничења захтевају рад са 64-битним типовима података. У C -у можемо користити тип *unsigned long long*, тј. вредности из сегмента $[0, 2^{64} - 1]$, што је управо нама и потребно. За рачунање вредности x , односно k -тог корена броја a , користимо функције:

- $\text{pow}(a, 1/k)$ у C -у
- $\text{exp}(\ln(a)/k)$ (ово је заправо ”вештачка” верзија функције степена јер користимо природни логаритам као међукорак, односно $\sqrt[k]{a} = a^{1/k} = e^{\ln(a^{1/k})} = e^{\frac{1}{k} \ln a}$)

Међутим као што смо напоменули, овде имамо један мали проблем. Наиме, при позиву функције pow која као параметар прима тип *double* може, за веће вредности улазних података, изгубити на тачности. Примера ради извршавањем следећег кода

```
unsigned long long a = 18446744073709551615uLL;
double tmp = (double) a;
printf (“%lf \n”, a);
```

на стандардном излазу ће бити исписано: 18446744073709552000.000000. Међутим, ову замку смо избегли, а да тога нисмо ни свесни. Наиме, у коду проверавамо да ли је $pow(x, k) < a$ и ако јесте повећавамо x за један. Овим додатним испитивањем сигурно добијамо тражену вредност x као цео број (наравно ово не важи за реалну вредност). Велики број такмичара је због овог превида, освојио 85 бода.

Сложеност овог алгорита је тешко тачно израчунати. Међутим, она је, могло би се рећи константна, јер у сваком од 63 корака имамо рачунање степена који може бити имплементиран у линеарном или (мада овде непотребно) логаритамском времену. У овом проблему је акценат био на прецизности, а не на брзини.

Напомена. Друга могућност је била имплементација посебне функције за рачунање k -тог корена. На овај начин пребацивање у *double* не би било потребно, тако да се не би губила горе описана прецизност. **Наведена функција се може имплементирати уз помоћ бинарне претраге.**

Производ цифара

Дат је цео број N . Исписати најмањи позитиван цео број X такав да је производ цифара броја X једнак броју N . Уколико такав број не постоји исписати -1 .

Улаз. (Улазни подаци се учитавају на стандардног улаза) Први ред стандардног улаза садржи број $0 \leq N \leq 10^{18}$.

Излаз. (Излазне подаци се исписују на стандардни излаз) У првом реду стандардног излаза исписати број X .

Пример 1.

<code>proizvod.in</code>	<code>proizvod.out</code>
56	78

Ограничења. Временско ограничење: 0.1s; Меморијско ограничење: 16MB

Решење и анализа. При анализи проблема наилазимо на два дела: да ли представљање у траженом облику постоји и ако постоји потребно је наћи најмањи такав број. Размотримо прво услове за егзистенцију решења.

Сваки природан број може бити представљен на јединствен начин као производ простих бројева, до на поредак (пошто је множење комутативна операција односно $a \cdot b = b \cdot a$). Овај диван резултат је позната као основна теорема алгебре. Како се од нас тражи да пронађемо број чији је производ цифара једнак датом број N то значи да се он може представити као производ једноцифрених простих бројева. Заиста, уколико би прост број $p > 10$ делио број N тада он не би могао бити изражен као производ бројева мањих од 10 односно производ цифара. Дакле, имамо да је N мора бити облика $2^a \cdot 3^b \cdot 5^c \cdot 7^d$ да би тражени број X постојао.

Како најлакше испитати ово? Ограничење броја N је 10^{18} , зато једноставним делењем редом са 2, 3, 5 односно 7, све док је то могуће долазимо до неке нове вредности N . На крају, број N мора завршити на јединицу. У супротном, тражено представљање не постоји и као резултат враћамо -1 .

У тексту проблема се захтева да број X буде најмањи могућ. Како ово постићи? Наиме, како посматрамо цифре број X прво што желимо да минимизујемо јесте управо број цифара. Како знамо њихов производ, минимални број цифара добијамо уколико за сваку од њих бирамо управо највећу могућу вредност. Други речима, минимални број цифара форме X добијамо тако што природни број N делимо са највећом цифром, односно са 9, све док је то могуће. Затим прелазимо на цифру 8 и све тако до цифре 2 (цифру 1 не треба посматрати, јер она само повећава број цифара, а не утиче на производ). На овај начин се добија минимални број цифара траженог број X .

На овај начин смо одредили минимални број цифара траженог броја X . Поставља се питање како наћи цифре овог броја. Цифре управо представљају описани делиоци при трежењу број цифара. Другим речима, ако смо број N поделили 5 пута цифром 9, тада ће тражени број X имати управо пет цифара 9 у свом запису. Оно што преостаје јесте да се ове цифре и сортирају. Сортирањем желимо да обезбедимо да цифре веће тежине имају мање вредности,

чиме добијамо најмању вредност броја X .

Горе описано сортирање је најбоље одрадити пребројавањем (енг. *count sort*). Другим речима, дефинишемо низ *count* дужине 9 и иницијализујмо вредности на нула. Када делимо број N цифром k , повећавамо бројач *count*[k] (он заправо представља неку врсту степена). На крају, сортирани низ цифара изгледа као: *count*[1] пута се појављује јединица, затим *count*[2] пута број два, све до *count*[9] пута број девет. Ово је најједноставнији алгоритам сортирања, међутим он се може применити само уколико је скуп вредности елемената низа мали и дискретан (као у ово случају где их има само 9).

Међутим, како ми редом и испитујемо цифре, нема потребе за сортирањем. Тачније, ми налазимо цифре почев од највеће, тако да је само потребно обрнути поредак. Ово можемо урадити на крају или при самој иницијализацији цифара.

Идеја

```
program ProizvodCifara;
var
  c: Integer;
  n: QWord;
  solution: String;
begin
  ReadLn(n);
  if (n = 0) then begin
    Writeln(10);
    Exit;
  end;
  if (n = 1) then begin
    Writeln(1);
    Exit;
  end;
  solution := '';
  for c := 9 downto 2 do begin
    while (n > 1) and (n MOD c = 0) do begin
      n := n DIV c;
      solution := c + solution;
    end;
  end;
  if (n <> 1) then
    Writeln(-1);
  else
    Writeln(solution);
end.
```

Менхетн

Међу датим скупом тачака $\{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$ исписати најмање менхетн растојање међу њима.

Менхетн растојање између тачака (a, b) и (c, d) се дефинише као $|a - c| + |b - d|$.

Улаз. (Улазни подаци се учитавају на стандардног улаза) У првом реду улаза налази се природни број n ($2 \leq n \leq 1000$). У наредних n редова се учитавају координате тачака: у $(k + 1)$ -ов реду се учитава тачка (x_k, y_k) ($-100.000 \leq x_k, y_k \leq 100.000$), где су x_k и y_k цели бројеви.

Излаз. (Излазне подаци се исписују на стандардни излаз) У првом и једином реду исписати најмање растојање.

Пример 1.

menhetn.in	menhetn.out
5	3
1 2	
4 2	
8 3	
9 5	
15 11	

Објашњење. Међу свим паровима тачака на најмањем растојању се налазе прва и друга ($|1 - 4| + |2 - 2| = 3$), и трећа и четврта тачка ($|8 - 9| + |3 - 5| = 3$).

Пример 2.

menhetn.in	menhetn.out
2	0
-1 1	
-1 1	

Ограничења. Временско ограничење: 0.1s; Меморијско ограничење: 16MB

Решење и анализа. Проблем Менхетн се показао као најлакши проблем на овим квалификацијама, где под најлакшим подразумевамо просечан број освојених бодова. Наиме, просечан број бодова на овом проблему је био 53 док је око 60% такмичара имало позитиван скор на њему.

Како је ограничење броја тачака 1.000, испитивање растојања сваког пара тачака посебно ће задовољити дато временско ограничења. Заиста, број могућих парова једнак је $\binom{n}{2}$ што је реда величине n^2 . Како је само рачунање растојања за дате две тачке константне сложености (само једна операција), сложеност овог алгорита је $O(n^2)$.

Означимо дати низ тачака са a . За имплементацију ове идеје потребна су нам два показивача i и j . Они ће показивати на тачке у низу a (горе описани пар тачака) које тренутно

испитујемо. Како је све једно да ли посматрамо тачке $a[i]$ и $a[j]$ или $a[j]$ и $a[i]$, можемо (без губљења општости) узети да је $i < j$. Једино на шта треба обратити пажњу јесте да индекси i и j морају бити различити (иначе би увек добили растојање 0 као решење). Због горње претпоставке да је $i < j$, индекс i треба "прошетати" од 1 до n а за j узимати вредности од $i + 1$ до n . Ово се имплементира угњежденим петњама по i и j .

У сваком кораку рачунамо растојање између тачака $a[i]$ и $a[j]$ (у псеудо коду означено са $currentDist$). Уколико је вредност овог растојања мања од вредности тренуног најмањег растојања ($minDist$ у псеудо коду) најмање растојања постављамо управо на ову вредност.

Алгоритам: Псеудо код проблема Менхетн

Input: низ тачака a дужине n

Output: најмање Менхетн растојања међу датим тачкама

```
minDist = ∞;
for i ← 1 to n do
  for j ← i + 1 to n do
    currentDist = |a[i].x - a[j].x| + |a[i].y - a[j].y|;
    if (currentDist < minDist) then
      minDist = currentDist;
    end
  end
end
return minDist
```

Напоменимо и то да је решење које испитује све парове $a[i]$ и $a[j]$, за које је $i \neq j$, коректно. Свако растојање би рачунали два пута, због симетрије, али то не би утицало на сложеност алгоритма. Међутим, уколико је неко убрзање лако увести (као у овом случају где је чак овај убрзани приступ једноставнији) свакако неће шкодити да исти имплементирате. У неким напреднијим проблемима ово може много значити.

У псеудо коду је иницијална вредност промењиве $minDist$ постављена на ∞ . Наравно, ово представља само нотацију у псеудо коду. Под бесконачношћу се подразумева било који број који је сигурно већи од свих могућих растојања међу овим тачкама. Како је ограничења за координате једнако 100.000 по апсолутној вредности, имамо да је максимално растојање које можемо очекивати једнако 400.000. Ово растојање се добија као растојање између тачака $(-x, -x)$ и (x, x) где је $x = 100.000$. Конкретан случај, као "најгори могући случај", је за дати проблем било једноставно наћи. То наравно није правило. Зато се увек треба оградити од евентуалне грешке постављањем ове вредности на већу (то свакако неће погоршати ствари).

Напомена. Интересантно је и напоменути зашто се овај проблем зове баш Менхетн (наравно ово није случајност). Наиме, овде смо рачунали растојања тачака у равни као апсолутно растојање по координатама (на први поглед неком делује чудно). Свима је вероватно познато стандардно растојање дефинисано као $\sqrt{(a.x - b.x)^2 + (a.y - b.y)^2}$ - познатије као Еуклидско растојање. Наиме, Еуклидско растојање представља најкраћи пут између две тачке у равни (баш зато што је праволинијски). Међутим, постоје и друге метрике (ово је формални израз за растојање над скупом тачака), а Менхетн је једна од њих. Други назив за ову метрику је Такси метрика (енг. *taxicab metric*) односно растојање. Назив потиче из начина кретања таксиста по блоковима града Менхетна који чине решетку (матрицу). Другим речима и ова метрика представља најкраће растојање између тачака али у "решетки".

Грешке такмичара. На крају, направимо кратак осврт на неке од грешака на које смо

наишли. Најфреквентнија грешка била је везана за тип промењливе који не обухвата ограничења координата дата у тексту проблема. Тако су многи, уместо максималних 100, освојили једва 30 бодова на овом задатку. Користите шири тип.

Мало мање учесталија грешка била је везана управо за иницијалну вредност минималног растојања тј. бесконачност о којој смо мало пре дискутовали. Многи су, не удубљујући се много у разматрање могућих вредности растојања, постављали $\infty = 200.000$ (вероватно посматрајући само позитивне координате). Горе наведена напомена, да се ова вредност увек "мало" повећа како би се избегли оваки и неки други специјални гранични случајеви, би решила овај проблем.

ДОДАТАК А: Структура *Trie*

Структура *Trie* представља једну од лакших структура података намењених индексирању и претрази стрингова. Већина структура овог типа нису подржане од стране стандардних библиотека.

Нека *word* представља конкретан стринг а са *dictionary* означимо скуп различитих речи. желимо да конструишемо такву 'структуру' којој ћемо моћи брзо испитати да ли одређена реч припада скупу или не. Постоје доста стандардних структура података које подржавају дати упит (нпр. *set*, *HashSet* ...). Међутим структуру *Trie* карактеришу две особине које наведене структуре немају:

- Убацавање нове речи као и испитивање припадности је сложености $O(|word|)$ (дакле не зависи од величине речника)
- Поред саме чињенице да ли реч припада скупу или не, из структуре *Trie* можемо пронаћи: речи чије је *edit-distance* растојање 1 (речи добијене заменом, брисањем или додавањем једног карактера); речи које почињу на дати префикс...

Назив *Trie* потиче из речи *retrieval* што значи претраживање. *Edward Fredkin* је 1960. године издао рад под називом '*Trie Memory*' у коме је описао ову структуру. Идеја која се крије иза самог назива, како је он рекао, се односи на повезивање две основне карактеристике структуре: облик стабла и функција претраге.

Као што смо већ напоменули, костур структуре ће бити представљен коренским стабло. Главна идеја структуре јесте да сам чвор не садржи у потпуности информацију, већ она налази на путу од њега до корена стабла. Дакле, за разлику од класичних структура које се репрезентују стаблом, сваки чвор ће као податак чувати само један карактер, док се сама реч коју он репрезентује добија конкатенирањем карактера на путу од корена до посматраног чвора.

Структуру можемо дефинисати на следећи начин:

Дата је скуп W различитих речи дефинисаних над коначном азбуком $\Sigma \cup \eta$, где η означава празан карактер. Коренско стабло T које задовољава следеће услове:

- сваки чвор стабла садржи два податка: карактер *data* и бинарну вредност *flag*
- подаци корена стабла су $data = \eta$ и $flag = 0$
- за сваки чвор стабла реч која је добијена конкатенирањем карактера *data* са чворова на пут од корена до њега, представља префикс неке речи из скупа W . Префикс ће представљати целу реч ако је вредност *flag*-а у чвору једнак 1
- за сваку реч $w \in W$ и сваки префикс p речи w постоји тачно један чвор чија је реч добијена горе описаним конкатенирањем управо префикс p
- чвор може имати највише једног директног потомка за дато $\varepsilon \in \Sigma$

назива се *Trie* стабло над речником W .

- *AddWord (string word)* - метода за додавање нове речи *word* у колекцију
- *Contains (string Word)* - метода за испитивање да ли реч *word* постоји у колекцији
- *Count (string prefix)* - функција за рачунање броја речи из колекције које садрже *prefix* као префикс

На почетку постављамо да је корен стабла *root* једини чвор у стаблу, при чему наравно постављамо његове атрибуте на полазне вредности. Додавање нове речи се извршава на следећи начин: крећемо од корена стабла и 'силазимо' у потомке који за *data* садрже одговарајући карактер. Уколико завршимо у чвору који нема таквог потомка, креирамо нови чвор са датим *data* и постављамо га као потомка чвора у коме се налазимо. чвору у коме смо завршили постављамо *flag* на 1.

Функција за додавање нове речи у колекцију

Input: Реч коју додајемо: *word*

```

currentNode = root;
for k ← 1 to length(word) do
    if currentNode нема потомка са data = word[k] then
        креирамо нови чвор child;
        child.data = word[k];
        child.flag = 0;
        поставити currentNode за родитеља чвора child;
    end
    currentNode = потомак чвора currentNode са data = word[k];
end
currentNode.flag = 1;

```

Испитивање припадности речи у колекцију је јако слично додавању нове. Крећемо се по стаблу почев од корена. Уколико дати потомак не постоји, реч не припада колекцију. У случају да само завршили обилазак свих карактера речи, једноставно треба испитати да ли је *flag* у чвору у коме смо завршили постављен или не.

Функција за испитивање да ли дата реч припада колекцији

Input: Реч коју испитујемо: *word*

Output: *true/false*: да ли реч припада колекцији или не

```

currentNode = root;
for k ← 1 to length(word) do
    if currentNode нема потомка са data = word[k] then
        | return false;
    end
    currentNode = потомак чвора currentNode са data = word[k];
end
if currentNode.flag = 0 then
    | return false;
end
return true;

```

На крају размотримо како се извршава функција *Count (string prefix)*. На почетку ћемо се кретати по стаблу преко карактера из префикса. Уколико је то немогуће, враћамо 0 као резултат. У супротном смо завршили у неком чвору стабла *v*. Резултат упита представља број чворова у подстаблу са кореном *v* који имају постављен бит *flag*. Ово можемо пребројати било којом методом претраге графа. Имплементација се оставља читаоцу за вежбу.
