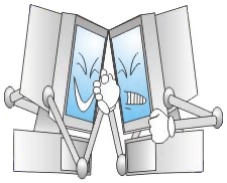


Прва рунда Квалификација за Окружно такмичење, 2012. године

Анализа проблема са решењима

Аутор: Андреја Илић
е-маил: andrejko.ilic@gmail.com

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. **упознавање 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, јако често ученици имају проблема са разумевањем концепта задатака који су заступљени на такмичењима из програмирања. На почетку сваког циклуса наилазимо на питања око учитавања / исписивања података (било са стандардног улаза / излаза било из датотека), такмичари нису упознати са системом тестирања, бодовањем проблема итд. Овим додатним рангом такмичења, на којем ученици имају недељу дана за пет проблема, желимо да многе проблеме овог типа избегнемо на окружном тамичењу. Такмичарима препоручујемо да прочитају текст *"Уобичајне грешке на националним такмичењима из програмирања"* и упознају се са неким "стандардним" стварима на које могу да се саплету. Текст можете преузети са сајта комисије, секција текстови, или директно преко линка: *http://www.yuoi.nis.edu.rs/tekstovi/09.02.2011/takmicarske_greske.pdf*.

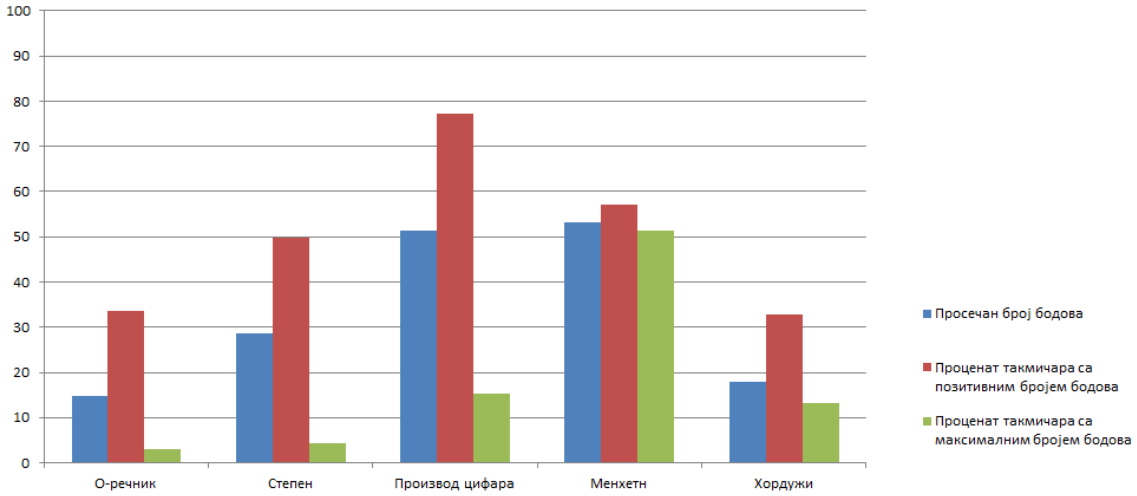
У овом документу су дата решања и анализе проблема са прве рунде Квалификација 2012. године. Описи идеја и алгоритама су јако детаљни, а као последица овога је дужина скрипе. Ипак, сигурни смо да ће решења, упркос дужини, бити занимљива (а надамо се и поучна) чак и онима који су исте проблеме и решили. На почетку је дат и кратак осврт на ово такмичење са статистикама проблема.

Уколико имате коментара или додатних сугестија поводом решења, немојте оклевати да се обратите било аутору било такмичарској комисији.

Кратка статистика такмичења

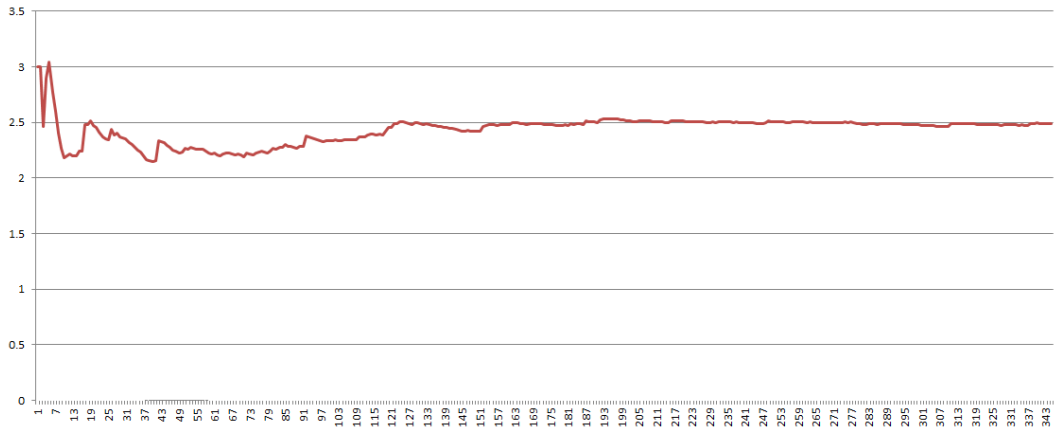
Прва рунда Квалификација за Окружно такмичење, за школску 2011/2012. годину, одржана је у периоду 23 - 30. јануара 2012. године. Као и прошлих година, такмичење је одржано на *z-trening* сајту. Учествовало је укупно 375 такмичара, од који је 314 било из Србије. Ово је најбољи одазив ученика од увођења овог такмичења.

Такмичари су имали период од недељу дана да се играју са 5 проблема. Тежине проблема су се кретале од елементарног па све до "републичког" нивоа. Више од 90% ученика је остварило позитиван резултат, док је само четворици такмичара пошло за руком да освоји максимални број бодова.



Графика 1. Статистика урађености проблема.

Посечан број *submit*-а био је око 7.5, што није велики број за 5 проблема. Међутим, просечан број *submit*-а по задатку, рачунајући само оне проблеме на којима је такмичар имао покушај, био је око 2.5. Ово генерално подржава чињеницу да такмичари мало времена посвећују тестирању својих имплементација.



Графика 2. График просечног броја *submit*-а по задатку у односу на број такмичара.

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.

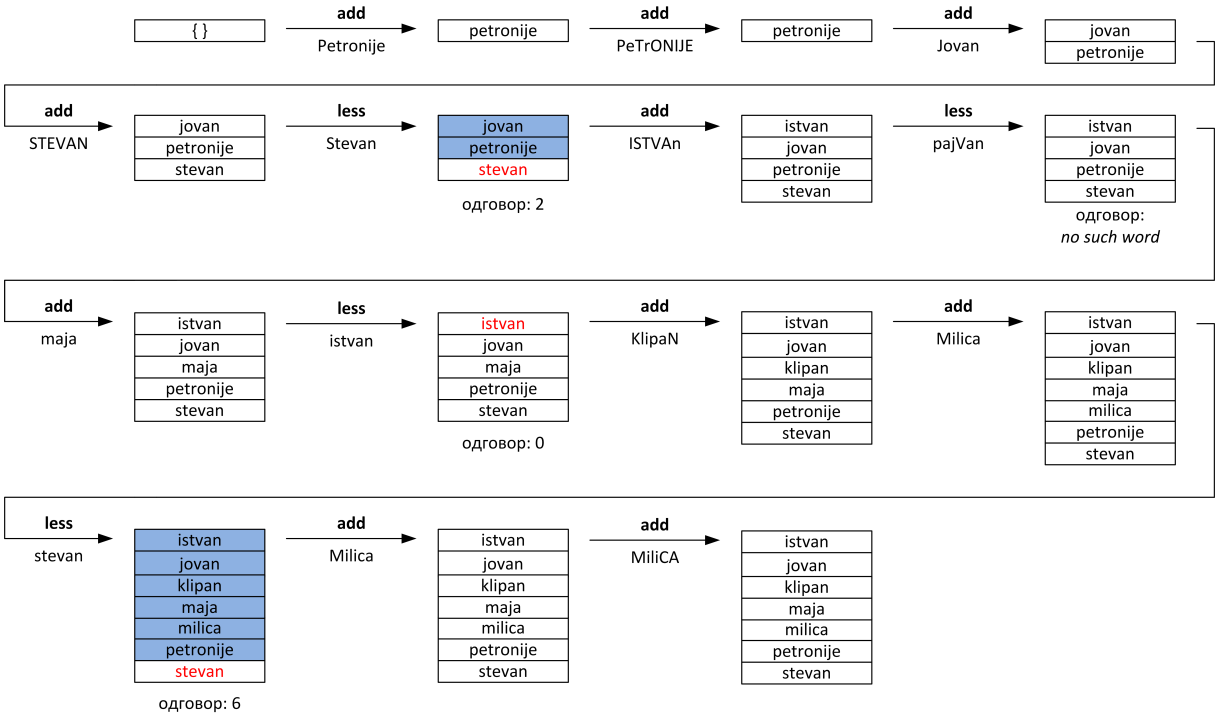
Ограничења. Временско ограничење: $1s$; Меморијско ограничење: $64MB$

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)

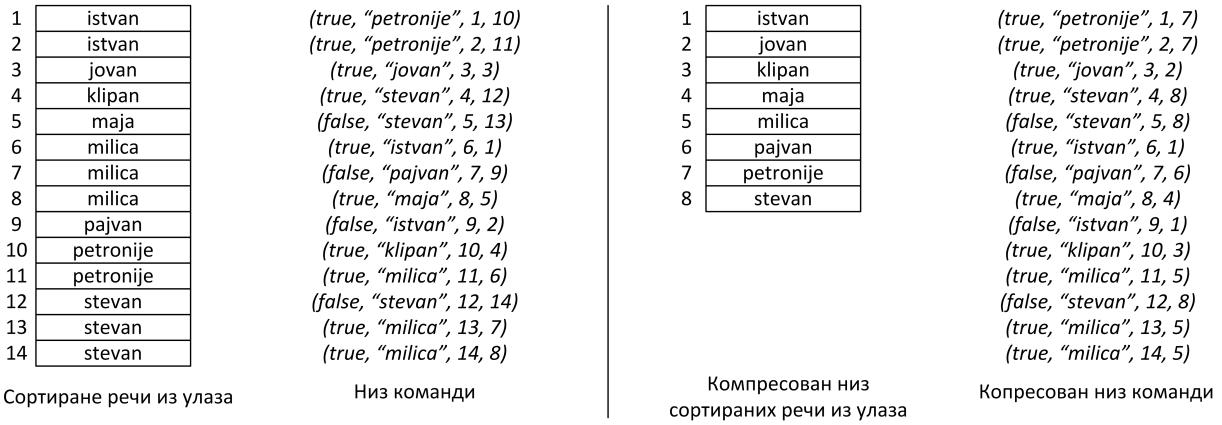
Дакле, при само учитавању креирамо низ команди и за сваку од њих иницијализујемо прва три атрибута. Овај низ команди, можемо сортирати лексикографски по атрибуту *word* односно по речима на које се команде односе. Шта добијамо овим? Овим се добија управо поредак речи на крају односно након извршења свих 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.

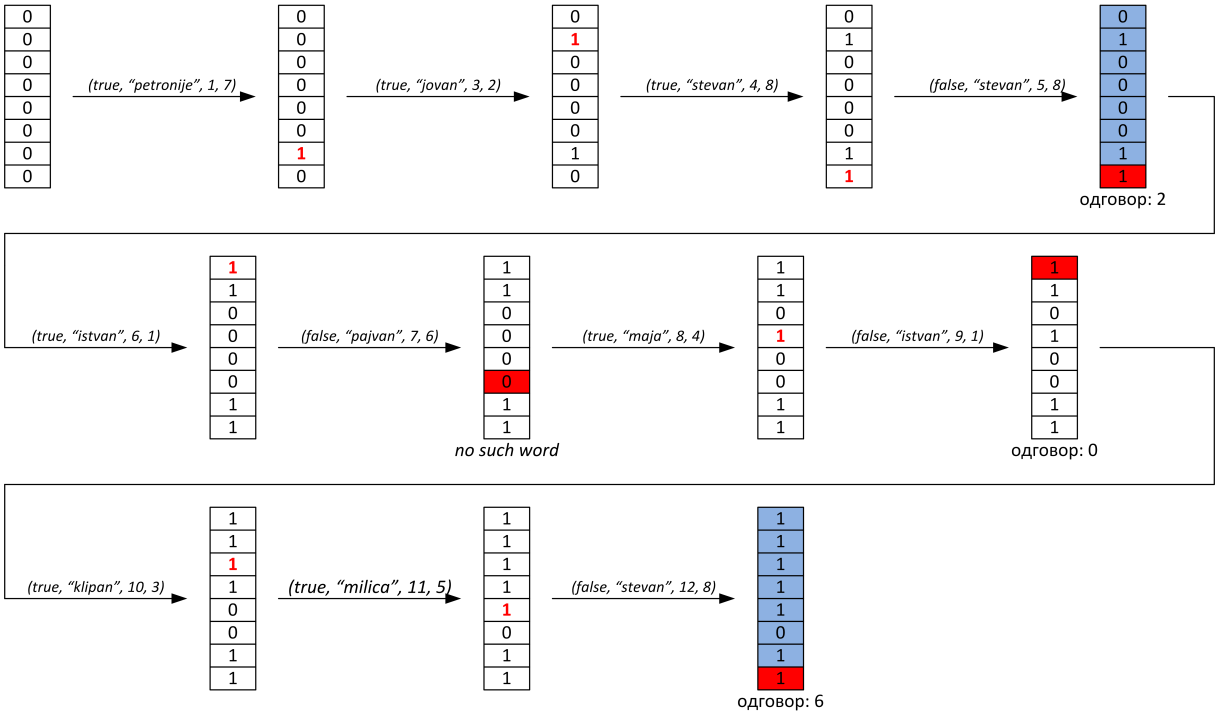
Проблем смо свели на симулацију следећег алгоритама:

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

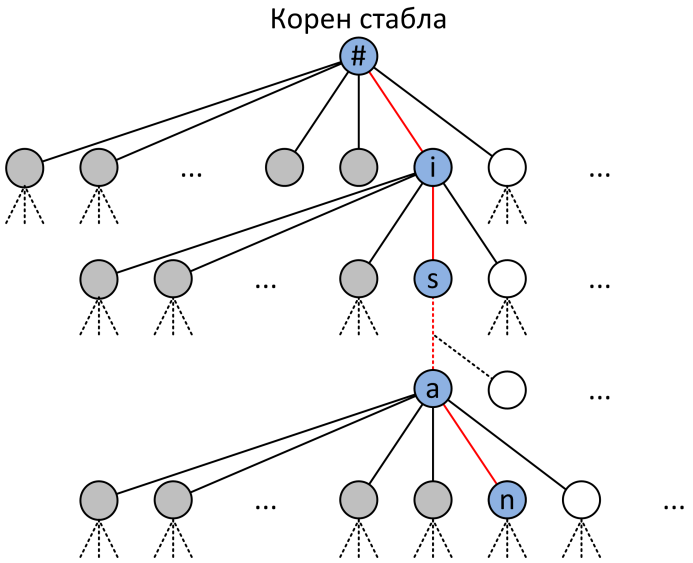
осталог), тако да се и њиховом имплементацијом добија иста сложеност. Наравно, препоручујемо да увек примењујете ону у чијем друштву се сигурније осећате.

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 сва деца 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.

Проблем 2. 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.

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

Пример 1.

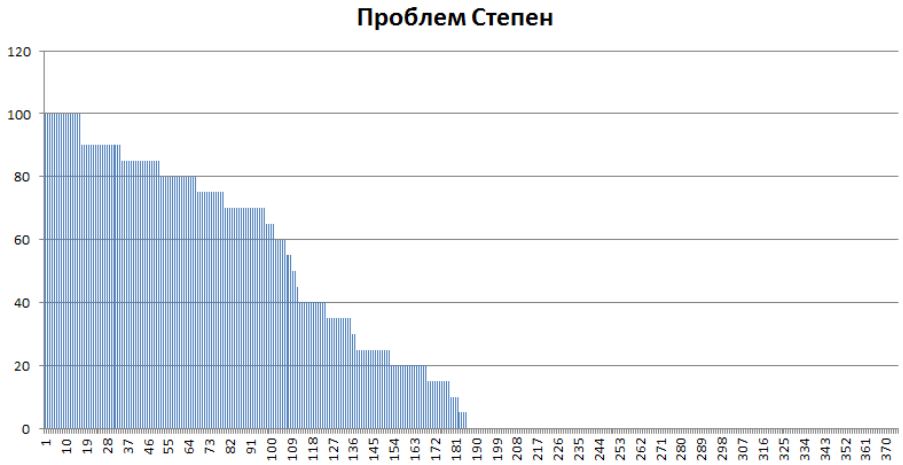
stepen.in	stepen.out
5 20	2 4

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 алгоритма. 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.



Слика 1. График броја освојених бодова свих такмичара.

Прва ствар коју треба приметити јесте да су могуће вредности степена k јако 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 $\{1, 2, \ldots, 63\}$.

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:

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

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 вреднити, 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*, а у *Pascal*-у тип *Qword*. Оба типа узимају 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:

- *pow (a, 1/k)* у *C*-у
- *Exp(ln(a) / k)* у *Pascal*-у (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 фунцкције *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: 184467440737095520000.000000. Међутим, ову замку смо избегли, а да тога нисмо ни свени. 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 алгоритма 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 импелементација 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 презицност. Наведена функција се може имплементирати уз помоћ бинарне претраге.

Проблем 3. 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 садрзи 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.

proizvod.in	proizvod.out
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 прозивод простих бројева, до на поредак (пошто је множење комутативна операција односно $a \cdot b = b \cdot a$). Овај диван резулат је позната као основна теорема алгебре. Како се од нас тражи да пронађемо број чији је производ цифара једнак датом броју 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 дељењем редом са $2, 3, 5$ односно 7, све док је то могуће долазим 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 броја 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 вресту 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.

Комплетан код у *Pascal*-у је дат у наставку.

```
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.
```

Проблем 4. Менхетн

Међу датим скупом тачака $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ 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 1 2 4 2 8 3 9 5 15 11	3

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 -1 1 -1 1	0

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 алгоритма 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 импелемтира 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
```

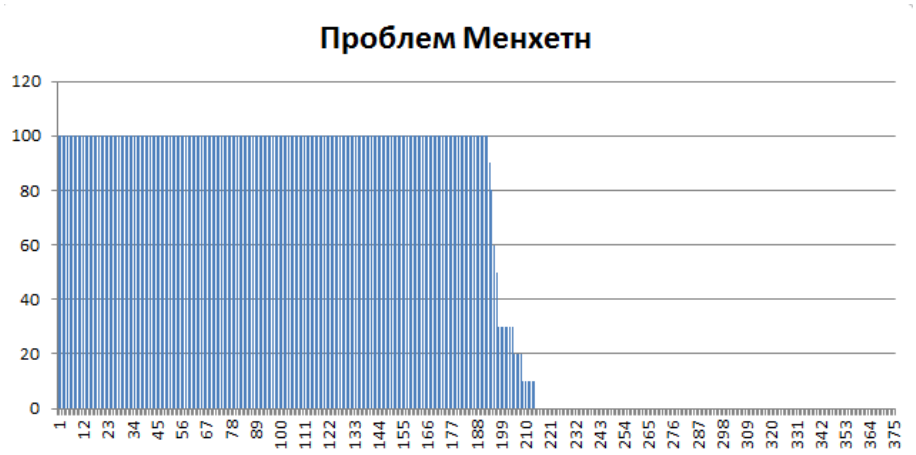
Напоменимо и то да је решење које испитује све парове $a[i]$ и $a[j]$, за које је $i \neq j$, коректно. Свако растојање би рачунали два пута, због симетрије, али то не би утицало на сложеност алгоритма. Међутим, уколико је неко убрзање лако увести (као у овом случају где је чак овај убрзнани приступ 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 координатана (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. *taxycab 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 промењиве. Наиме, многи такмичари који су кућали у *Pascal*-у су за тип узимали *Integer* уместо *LongInt*. Међутим, овај тип узима вредности из сегмента $[-2^{15}, 2^{15}-1]$ који не обухвата ограничења координата дата у тексту проблема. Тако су многи, уместо максималних 100, освојили једва 30 бодова на овом задатку.



Слика 1. График броја освојених бодова свих такмичара.

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.

Проблем 5. ХорДужи

Перица је на папиру нацртао n хоризонталних дужи. Затим је тај папир дао свом другу Јовици и задао му задатак да преброји све дужи. Међутим, ако се неке дужи преклапају, Јовица неће приметити да су то различите дужи, већ ће их посматрати као једну дуж. Две дужи се преклапају ако имају бар једну заједничку тачку.

Колико дужи ће Јовица да преброји?

Улаз. (Улазни подаци се учитавају на стандардног улаза) Први ред стандардног улаза садржи један природан број n ($1 \leq n \leq 100.000$), број дужи. У следећих n редова, налазе се по три цела броја y_i , xa_i , xb_i ($-1.000.000.000 \leq y_i, xa_i, xb_i \leq 1.000.000.000$) који означавају да крајеви i-те дужи имају координате (xa_i, y_i) и (xb_i, y_i).

Излаз. (Излазне подаци се исписују на стандардни излаз) У први и једини ред стандардног излаза исписати колико дужи види Јовица.

Пример 1.

horduzi.in	horduzi.out
7 7 5 8 3 8 4 4 1 2 3 5 2 3 3 10 7 2 5 7 12 9	4

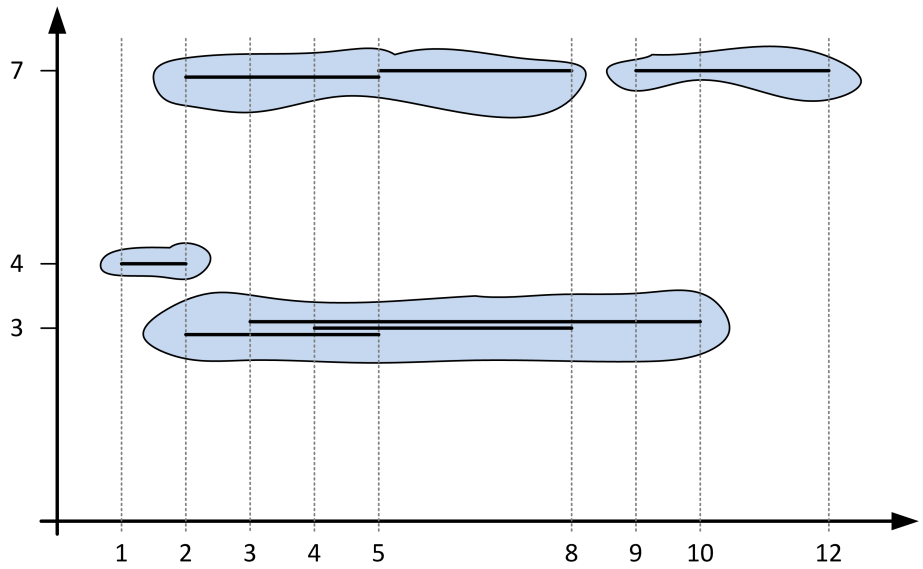
Ограничења. Временско ограничење: 1s; Меморијско ограничење: 16MB

Решење и анализа. Идеја овог проблема је позната и постоји доста варијанти исте (неке од њих ћемо напоменути касније). Међутим, на квалификацијама се овај проблем показао као солидно чврст орах - просечан број бодова био је свега око 18, док око две трећине такмичара нема позитиван број бодова на њему.

Анализу проблема ћемо започети анализом датог примера (што је увек добар почетак при решавању). На почетку напоменимо да границе дужи у улазу нису сортиране (xa не мора бити мање од xb). У даљем делу анализе претпостављамо да важи поредак $xa < xb$, тј. да смо ове вредности заменили на самом улазу уколико поредак није био добар. Пример можемо приказати графички као на слици 1, при чему ово није баш реална ситуација - одређене дужи су мало спуштене или подигнуте како би се приметила места преклапања. Решење, односно дужи које Јовица види су означена у плавим облацима.

Ради лакшег описа алгоритма, скуп дужи које Јовица види као једну називамо групом. Прво што можемо приметити, а што се дало и наслутити, јесте да се дужи са различитом y координатом никада не могу наћи у истој групи односно преклопити у некој тачки. Дакле, за свако y проблем можемо посматрати независно. Ово је битна и јако лепа чињеница јер смо на основу ње проблем свели на дужи које се налазе на "истој висини". Како наћи дужи које

су на истој висини? Тривијално решење, које наравно није добро, је да за сваку вредност y координате тражимо дужи на њој. Међутим, бољи приступ је да се дужи сортирају по y координатама. Ова идеја се сама намеће, јер на тај начин добијамо да се све дужи на истом нивоу (иста y координата) налазе једна уз другу. Кокретно, скуп дужи са истог нивоа представља подниз сортираног низа.



Слика 1. Графички приказ примера "са папира".

Сортирањем дужи из примера добија се низ:

$$(3,2,8),(3,3,10),(3,4,8),(4,1,2),(7,2,5),(7,5,8),(7,9,12)$$

где примећујемо да су дужи са $y = 3$ координатом од прве до треће, да је дуж са индексом 4 једина са $y = 4$ координатом, док су дужи на нивоу 7 од пете до седме. Сваки од ових скупова дужи на истом нивоу посматрамо посебно. Овим смо добили нови подпроблем:

за дате дужи на x-оси (односно истом нивоу), наћи број дужи које Јовица види

Дакле, сада можемо посматрати дужи са крајевима $(a_1, b_1), (a_2, b_2), \ldots, (a_m, b_m)$. Посматрајмо дужи (a_i, b_i) и (a_j, b_j) и претпоставимо да оне припадају једно групи дужи. Ако се оне секу, добијамо да је дуж коју оне граде, односно њихова унија, заправо $(\min\{a_i, a_j\}, \max\{b_i, b_j\})$. Примера ради, дужи $(3, 5)$ и $(4, 6)$ које се секу граде дуж $(3, 6)$. Како је потребно наћи ове "ланце" дужи које се преклапају или додирују, некако сортирање опет виси у ваздуху као кључни корак. Идеју методе коју ћемо овде изнети је теже објаснити овако "с неба па у ребра". Зато ћемо одмах почети са описом методе а на крају продискутовати о идеји.

Дефинишимо нови низ x и паралелно са њим низ *open* на следећи начин:

$$x[2k+1] = a_k,\ open[2k+1] = 1$$

$$x[2k] = b_k,\ open[2k] = -1$$

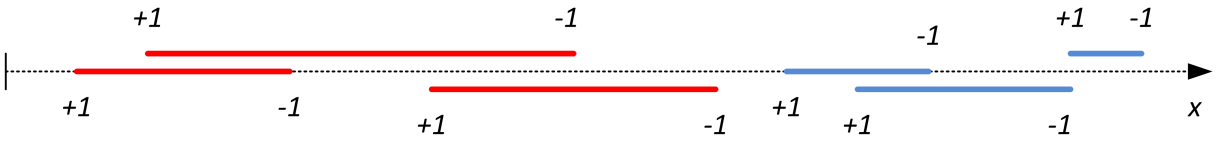
Другим речима, у низ x редом ређамо крајеве дужи а у низу *open* плус и минус један. Као што видимо 1 означава почетак дужи а -1 њен крај. Сада долази на реду сортирање. Наиме,

сортирамо низ x у неопадајући поредак, а паралелно са њим мењамо и низ *open* (ово можемо посматрати и као структуру која има x координату и параметар *open* док се сортирање врши упоређивањем само атрибута x). Специјалан случај је када се две дужи секу у једној тачки односно крај прве је почетак друге. Тада у низу x имамо два исте вредности али једној одговара $+1$ а другој -1 у низу *open*. Зато уводимо правило сотирања да уколико су x координате једнаке тада се сортира по *open* али тако да прво иде $+1$ а затим -1 (другим речима нерастуће по *open*). Касније ћемо видети зашто је ово битно (односно шта се мења уколико се сортира само по x координати).

Шта смо добили на овај начин? Ако се крећемо по низу x ми се заправо крећемо по крајевима датих дужи али у неопадајућем поретку. Крајеве дужи ћемо ословљавати са почетак и крај, респективно. Уколико наиђемо на почетак неке дужи, тада све дужи на чији почетак наиђемо пре затварања ове дужи, припадају истој групи. Дакле уколико постоји нека дуж која још увек није затворена (нисмо још увек наишли на њен крај) тада све дужи на које наилазимо припадају истој групи као и та не затворена дуж. У нашем случају није битно које дужи припадају којим групама, тако да последњу реченицу можемо превести како: крај неке дужи је и крај групе ако и само ако су све дужи пре ње затворене.

Ово нас наводи на следећи алгоритам:

- Иницијализујмо број група и број отворених дужи на нула, односно $numGroups = 0$ и $numOpen = 0$
- Редом се крећемо по низу x, а самим тим и по низу *open*. Означимо тренутни индекс у овим низовима са k.
  - Уколико смо наишли на почетак неке дужи, односно $open[k] = 1$, повећавамо број отворених дужи за 1 тј. $numOpen = numOpen + 1$.
  - Уколико смо наишли на крај неке дужи, односно $open[k] = -1$, број отворених дужи смањујемо за један тј. $numOpen = numOpen - 1$. Ако је ово смањивање довело до тога да је $numOpen = 0$, тада смо затворили текућу групу и укупан број група повећавамо: $numGropus = numGroups + 1$.



Слика 2. Пример за вредности *open* низа. Након сортирања добија се низ $open = (1, 1, -1, 1, -1, -1, 1, 1, -1, 1, -1, -1)$.

Зашто смо на почетку изабрали баш вредности $+1$ и -1 за елементе низа *open*? Логичније би било да смо за то искористили неки *boolean* низ (с обзиром да су нам потребне само две вредности) чиме штедимо меморију. Међутим, шта примећујемо у опису алгоритма: када је $open[k] = 1$ број отворених дужи се повећава за један, а када је негативан тада се смањује за један. Сада видимо да за ово не морамо разликовати случајеве, једноставно можемо записати само $numOpen = numOpen + open[k]$. Ово је свакако елегантније од гранања.

Овде налазимо и одговор на питање сортирања: зашто сортирамо по x, а уколико су једнаки онда по *open*? Наиме, уколико би неко затварање било пре отварања, тада би *numOpen* могао да постане нула и пре затварања групе. Примера ради, уколико имамо само две дужи $(1, 2)$ и $(2, 3)$, тада би имали да је (у овом погрешном резоновању): $x = (1, 2, 3, 4)$ и $open = (1, -1, 1, -1)$.

Сада би имали да *numOpen* узима вредности $1, 0, 1, 0$, редом. Како два пута узима вредност нула, добили би да је укупан број група 2, а заправо је само један. Зато низ *open* мора бити сортиран као: $open = (1, 1, -1, -1)$.

Вратимо се на наш почетни проблем. Да ли ми морамо посматрати све скупове дужи са истим y координатама засебно? Не морамо, јер ће свакако након завршетка дужи из једног нивоа, број отворених дужи бити нула, тако да можемо директно наставити са другим скупом дужи. Како је тај други скуп дужи одмах у наставку низа (подсетимо се да су дужи сортиране прво по y) једноставно ова подела по скуповима се имплицитно решава сортирањем.

Када сву ову причу спојимо, добијамо да је алгоритам који решава овај проблем доста једноставан за имплементирање (иако то на први поглед не делује тако). Наиме, дефинишимо структуру са атрибутима $(x, y, open)$. За сваку дуж са улаза, у низ p ових струкутра убацујемо $(xa_i, y_i, 1)$ и $(xb_i, y_i, -1)$. Затим сортирамо овај низ прво по y, за једнаке y по x у неопадајући поредак, а за једнаке и x и y по *open* (монотоност при сортирању по y није битна). Након тога се крећемо по низу и стално инкрементирамо вредност *numOpen* за вредност *open* из структуре и бројимо колико пута узима вредност нула.

Како дефинисати критеријум сортирања? Наиме, као што смо видели, низ чији су елементи описане структуре треба сортирати по три критеријума. Означимо са A и B две инстанце ове структуре. Тада критеријум сортирања можемо описати као:

$$(A.y < B.y) \vee ((A.y == B.y) \wedge (A.x < B.x)) \vee ((A.y == B.y) \wedge (A.x == B.x) \wedge (A.open > B.open))$$

Главни део имплементације (без учитавања тј. инизијализације низа p) се може искодирати у само петнаестак линија кода:

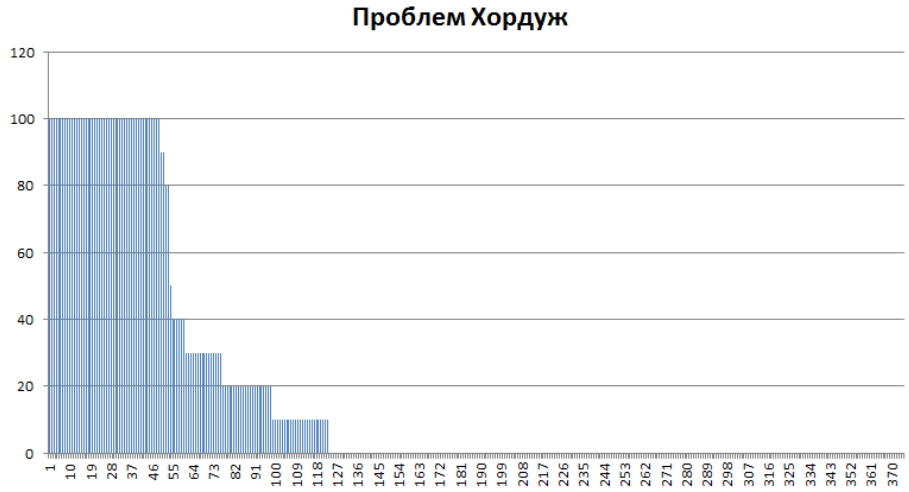
```
int compare (const void * a, const void * b)
{
    POINT A = *(POINT *)a;
    POINT B = *(POINT *)b;
    if ((A.y > B.y) || ((A.y == B.y) && (A.x > B.x)) || ((A.y == B.y) && (A.x == B.x) && (A.open < B.open)))
        return 1;
    return -1;
}

int solve()
{
    int toReturn = 0;

    qsort (p, 2 * n, sizeof(POINT), compare);

    int numOpen = 0;
    for (int i = 0; i < 2 * n; i++)
    {
        numOpen = numOpen + p [i].open;
        if (numOpen == 0)
            toReturn++;
    }

    return toReturn;
}
```

Сложеност изложеног алгоритма је $O(n \log n)$. Заиста, након сортирања потребан је само један пролазак кроз низ. Меморијска сложеност је линеарна по n.

Напомена. Постоје многе варијанте овог проблема. Неке од њих су (проблеми су изложени у једној димензији):

- за дате дужи на x-оси израчунати укупну дужину које оне прекривају;

- за дате дужи на x-оси наћи тачку која припада највећем броју дужи (решење ове варијанте је максимална вредност коју узима *numOpen*);
- за дате дужи на x-оси наћи оне дужи које припадају истој групи као и дата дуж d;



Слика 3. График броја освојених бодова свих такмичара.

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 стриг а са *dinctionary* 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:

- Убацивање 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 *retrival* 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 већнапоменули, костур структуре ће бити представљен коренским стабло. Главна идеја стрктуре јесте да сам чвор не садржи у потпуности инфромацију, већсе она налази на путу од њега до корена стабла. Дакле, за разлику од класичних струкутра 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 акко 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.

Иако формално делује компликовано, конструкција стабла је врло једноставна. Пре него што почемо са имплементацијом, размотрићемо *Trie* стабло добијено над суфиксима 6-те Фибоначијеве речи[1] *abaababa*. Дакле имамо да је

$$W = \{a, ba, aba, baba, ababa, aababa, baababa, abaababa\}$$

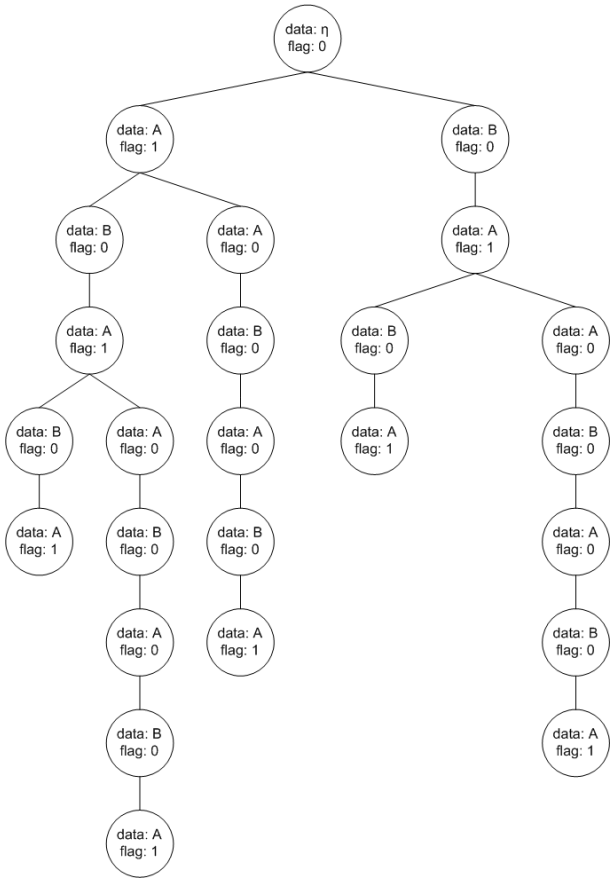


Figure 1: Trie nad sufiksima 6-te Fibonachijeve rechi

Означимо са $word(v)$, где је v чвор стабла, реч добијену конкатенирањем карактера са пута од корена стабла до самог чвора. Посматрајмо сада произвољан чвор v стабла, који је различит од корена. Тада имамо да $word(v)$ представља префикс неке речи из скупа W. Тачније број речи које имају $word(v)$ као префикс једнак је број чворова из подстабла чвора v (подстабла чији је корен v) код којих је $flag$ једнак 1. Такође видимо да су сви листови имају постављен $flag$ бит, јер они сигурно представљају крај неких речи из речника W.

Имплементација

У овом делу ћемо изложити имплементацију основне верзије структуре *Trie*, којом испитујемо егзактно поклапање. Алгоритам ће бити представљен псеудо кодом. Наиме, структуру описујемо помоћу три основне функције:

[1] Фибоначијеве речи се добијају рекурзивним алгоритмом сличним самој констркцији Фибоначијевих бројева. Дефинишемо их као: $Fib_1 = a$, $Fib_2 = b$, док је $Fib_n = Fib_{n-1}Fib_{n-2}$ за $n > 2$. Фибоначијеве речи су интересантне јер имају велики број карактеристика периодничности и понављања.

- *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 currentNodeнема потомка са data = word[k] then
        креирамо нови чвор child;
        child.data = word[k];
        child.flag = 0;
        поставити currentNodeза родитеља чвора child;
    end
    currentNode = потомак чвора currentNodeса 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/falase: да ли реч припада колекцији или не
currentNode = root;
for k ← 1 to length(word) do
    if currentNode nema potomka sa data = word[k] then
        return false;
    end
    currentNode =  potomak chvora currentNode sa 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.