65. ročník Matematické olympiády – 2015/2016

Úlohy ústředního kola kategorie P – 1. soutěžní den

Na řešení úloh máte 4,5 hodiny čistého času. Řešení každé úlohy pište na samostatný list papíru. Při soutěži je zakázáno používat jakékoliv pomůcky kromě psacích potřeb (tzn. knihy, kalkulačky, mobily, apod.).

Řešení každého příkladu musí obsahovat:

- **Popis řešení,** to znamená slovní popis principu zvoleného algoritmu, argumenty zdůvodňující jeho správnost, diskusi o efektivitě vašeho řešení (časová a paměťová složitost). Slovní popis řešení musí být jasný a srozumitelný i bez nahlédnutí do samotného zápisu algoritmu (do programu). Není povoleno odkazovat se na Vaše řešení předchozích kol, opravovatelé je nemají k dispozici; na autorská řešení se odkazovat můžete.
- **Zápis algoritmu.** V úlohách **P-III-1** a **P-III-2** je třeba uvést zápis algoritmu, a to buď ve tvaru zdrojového textu nejdůležitějších částí programu v jazyce Pascal nebo C/C++, nebo v nějakém dostatečně srozumitelném pseudokódu. Nemusíte detailně popisovat jednoduché operace jako vstupy, výstupy, implementaci jednoduchých matematických vztahů, vyhledávání v poli, třídění apod. V řešení úlohy **P-III-3** využijte sufixové stromy takovým způsobem, jak je ukázáno ve studijním textu; nemusíte se starat o jejich implementaci.

Za každou úlohu můžete získat maximálně 10 bodů. Hodnotí se nejen správnost řešení, ale také kvalita jeho popisu (včetně zdůvodnění správnosti) a efektivita zvoleného algoritmu. Algoritmy posuzujeme zejména podle jejich časové složitosti, tzn. podle závislosti doby výpočtu na velikosti vstupních dat. Záleží přitom pouze na řádové rychlosti růstu této funkce.

V zadání každé úlohy najdete přibližné limity na velikost vstupních dat. Efektivním vyřešením úlohy rozumíme to, že váš program spuštěný s takovými daty na současném běžném počítači dokončí výpočet během několika sekund.

P-III-1 Trojnohý běh

Čtvrťáci se už těší na příští pátek. Místo vyučování totiž mají sportovní den. Nejsledovanější disciplínou je trojnohý běh. V trojnohém běhu mezi sebou soutěží dvojice chlapec-dívka. Při běhu je v každé dvojici pravá noha chlapce přivázána k levé noze dívky, takže oba musí běžet stejnou rychlostí.

Soutěžní úloha

Ve třídě je n chlapců a n dívek. O každém z nich víme, jakou má rychlost. Na závod v trojnohému běhu z nich chceme sestavit n dvojic. Rychlost dvojice je určena nižší z rychlostí obou dětí, které tuto dvojici tvoří.

Úkol A: *(5 bodů)* Napište co nejefektivnější program, který zjistí, jaký *nejvyšší* může být součet rychlostí všech sestavených dvojic.

Úkol B: *(5 bodů)* Napište co nejefektivnější program, který zjistí, jaký *nejnižší* může být součet rychlostí všech sestavených dvojic.

V obou úkolech uveďte *zdůvodnění správnosti* použitého algoritmu.

Formát vstupu a výstupu

První řádek vstupu obsahuje číslo n. Na druhém řádku je n čísel, která představují rychlosti chlapců. Na třetím řádku je také n čísel, což jsou rychlosti dívek.

Program vypíše dva řádky. Na prvním z nich je uveden nejvyšší a na druhém nejnižší možný součet rychlostí všech dvojic.

Příklad

Vstup:	*Výstup:*
3	9
5 1 4	6
2 3 10	

Nejvyššího součtu rychlostí dosáhneme například tím, že chlapec s rychlostí 5 poběží s dívkou s rychlostí 10, chlapec s rychlostí 1 s dívkou s rychlostí 2 a chlapec s rychlostí 4 s dívkou s rychlostí 3. Rychlosti těchto dvojic jsou 5, 1 a 3, součet všech rychlostí je tedy 9.

Nejnižší součet rychlostí dostaneme například pro dvojice 5–2, 4–3 a 1–10. Rychlosti těchto dvojic jsou 2, 3 a 1, jejich součet je 6.

P-III-2 Elektromobil

Honza si nedávno koupil nové moderní auto na elektřinu. Auto jezdí výborně, ale Honza s ním má jeden problém: je zde poměrně málo dobíjecích stanic. Dostat se z místa na místo tak může být docela obtížné.

Soutěžní úloha

V zemi je n měst a m silnic. Každá silnice je obousměrná a spojuje některá dvě města. Pro jednoduchost budeme předpokládat, že všechny silnice mají stejnou délku. V d městech jsou postaveny dobíjecí stanice. Auto má baterii, jejíž kapacita vystačí na projetí k silnic.

Pro zadaná dvě města a a b určete, zda je možné dojet autem do města b, pokud cestu začínáme ve městě a s plně nabitou baterií.

Formát vstupu a výstupu

Na prvním řádku vstupu jsou uvedena čísla n, m, d, k, a, b. Města jsou očíslována od 0 do $n - 1$. Na druhém řádku je d čísel, která představují čísla měst, v nichž jsou dobíjecí stanice. Každý z následujících m řádků popisuje jednu silnici – obsahuje čísla dvou měst, mezi nimiž silnice vede.

Na výstup vypište `ano` nebo `ne` podle toho, zda můžeme dojet autem z města a do města b.

Omezení a hodnocení

Plný počet bodů dostanete za řešení, které efektivně vyřeší všechny vstupy, v nichž $n \leq 100\,000$ a $m \leq 500\,000$ (přitom hodnoty d a k mohou být libovolné).

Poměrně velký počet bodů obdrží také řešení, které si poradí s podobně velkými hodnotami n a m, ale pouze za předpokladu, že některý z parametrů d nebo k je malý.

Libovolné správné řešení pracující v polynomiálním čase vzhledem k n a m může získat až 4 body.

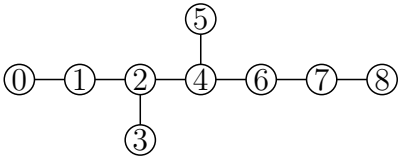
Příklad

Vstup:

```
9 8 2 3 0 7
3 5
0 1
1 2
2 3
2 4
4 5
4 6
6 7
7 8
```

Výstup:

```
ano
```



Pojedeme z 0 přes 1 a 2 do 3, kde dobijeme auto.
Odtud pojedeme přes 2 a 4 do 5, kde znovu dobijeme auto.
Nakonec pojedeme z 5 přes 4 a 6 do 7.
Pokud by na vstupu bylo zadáno $b = 8$ namísto $b = 7$, program by odpověděl `ne`.

P-III-3 Sufixové stromy

K této úloze se vztahuje studijní text uvedený na následujících stranách. Studijní text je identický se studijním textem z domácího a krajského kola.

Úkol A: *(4 body)* Určitě znáte osmisměrku – logickou úlohu, v níž je úkolem vyhledávat slova v zadané tabulce písmen. Možná jste si ale neuvědomili, že množina slov, která je třeba v osmisměrce vyškrtat, nemůže být úplně libovolná. Kromě jiného musí platit, že žádné z vyškrtávaných slov není podřetězcem jiného vyškrtávaného slova. Například v dobré osmisměrce nebudou zároveň slova `tvor` a `potvora`. Potom by se totiž mohlo stát, že nejprve najdeme a vyškrtneme slovo `tvor`, ale později se ukáže, že to nebyl ten správný `tvor`, ale část delšího slova `potvora`.

Na vstupu jsou zadány řetězce $S_1, \ldots, S_n$. Napište program s optimální časovou složitostí, který zjistí, zda je některý z těchto řetězců podřetězcem některého jiného.

Úkol B: *(6 bodů)* Cyklickým posunem řetězce rozumíme každý řetězec, který získáme tak, že původní řetězec rozdělíme na dvě části a druhou přesuneme před první. Například všechny cyklické posuny řetězce `zabka` jsou `zabka`, `abkaz`, `bkaza`, `kazab` a `azabk`. Pokud tyto cyklické posuny uspořádáme podle abecedy, dostaneme následující pořadí: `abkaz`, `azabk`, `bkaza`, `kazab`, `zabka`.

Je zadán řetězec S délky n a číslo k, pro které platí $1 \leq k \leq n$. Napište program s optimální časovou složitostí, který vypíše k-tý nejmenší cyklický posun řetězce S. Například pro $S =$ `zabka` a $k = 4$ bude správným výstupem řetězec `kazab`. Pro $S =$ `baba` bude pro $k = 1$ a také pro $k = 2$ správným výstupem řetězec `abab`.

Studijní text

V tomto studijním textu se seznámíme s jednou užitečnou datovou strukturou pro práci se znakovými řetězci: *sufixovým stromem*. Dozvíte se, jak tento strom *vypadá*. Nedozvíte se, jak takový strom *efektivně sestrojit* – ale to při řešení soutěžních úloh nebudete potřebovat. Úplně vám bude stačit, když dokážete tento strom *použít jako nástroj* při návrhu nových algoritmů.

Dříve, než se dostaneme k samotným sufixovým stromům, zavedeme si některé užitečné pojmy.

Abeceda

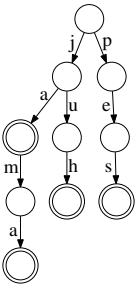
Vstupem všech soutěžních úloh budou znakové řetězce tvořené malými písmeny anglické abecedy. Kromě nich se nám občas bude hodit použít pracovně i některé další symboly. Budeme ale předpokládat, že všechny použité znaky mají ASCII hodnoty z rozmezí od 33 do 126. Velikost abecedy proto můžeme považovat za konstantní a nebudeme ji uvažovat při odhadech časové složitosti.

Písmenkový strom

Písmenkový strom (anglicky *trie*) je jednoduchá datová struktura, kterou můžeme použít pro uložení množiny řetězců. Je to zakořeněný strom, v němž platí:

- Každá hrana má přiřazeno jedno písmeno.
- Pro každý vrchol platí, že z něho vedoucí hrany mají navzájem různá písmena.
- Některé vrcholy jsou označeny.

Každému vrcholu v písmenkovém stromu odpovídá řetězec tvořený posloupností písmen, která přečteme na hranách stromu cestou z kořene do dotyčného vrcholu. Písmenkový strom představuje množinu těch řetězců, které odpovídají označeným vrcholům.



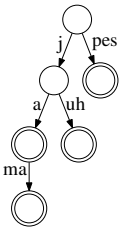
Písmenkový strom představující množinu řetězců {`ja`, `jama`, `juh`, `pes`}. *Označené vrcholy jsou znázorněny dvojitým kroužkem.*

Písmenkový strom reprezentující danou množinu řetězců dokážeme snadno sestrojit v čase přímo úměrném součtu jejich délek. Začneme s prázdným stromem, který je tvořen pouze neoznačeným kořenem. Postupně do stromu přidáváme jednotlivé řetězce. Přidání jednoho řetězce vypadá tak, že se z kořene stromu vydáme

dolů po cestě, která je určena znaky tvořícími řetězec. Několik našich prvních kroků může vést přes již existující vrcholy, následně budeme nuceni několik nových vrcholů a hran do stromu přidat. Nakonec ještě označíme ten vrchol, v němž jsme naši cestu zakončili.

Komprimovaný písmenkový strom

Písmenkový strom často zabírá zbytečně mnoho paměti. V každém vrcholu v si totiž musíme pamatovat pro každé písmeno x abecedy, zda a kam vede z v hrana označená x. Zlepšení lze dosáhnout kompresí hran. Jednoduše vynecháme ty vrcholy, kde se nic neděje – tedy neoznačené vrcholy, v nichž se písmenkový strom nevětví. V komprimovaném písmenkovém stromu tedy platí, že každá hrana má přiřazen neprázdný *řetězec*. Následně pro každý vrchol platí, že hrany z něj vedoucí mají navzájem různá *první* písmena.



Komprimovaná verze písmenkového stromu z předcházejícího obrázku.

Komprimovanou verzi písmenkového stromu dokážeme sestrojit podobně jako tu původní, jenom implementace je o něco složitější. Kdybychom například do stromu na obrázku chtěli přidat nový řetězec `pluh`, museli bychom současnou hranu označenou `pes` rozdělit novým vrcholem v na dvě kratší: hranu označenou `p` vedoucí z kořene do v, a hranu označenou `es` vedoucí z v dále. Následně bychom z v přidali druhou hranu označenou `luh`.

Prefixy, sufixy a podřetězce

Ve více úlohách se budeme zabývat podřetězci daného řetězce. Slovem *podřetězec* budeme vždy rozumět souvislý podřetězec, tedy úsek po sobě následujících písmen v původním řetězci. Tedy například řetězec `ace` není podřetězcem řetězce `abcde`.

Podřetězce začínající na začátku řetězce nazýváme *prefixy* a podřetězce končící na jeho konci nazýváme *sufixy*. Například řetězec `abcde` má sufixy `abcde`, `bcde`, `cde`, `de` a `e`. (Někdy za sufix považujeme i prázdný řetězec – tedy sufix nulové délky.)

Všimněte si užitečné vlastnosti: ať si zvolíme jakýkoliv podřetězec daného řetězce, vždy existuje sufix, který tímto podřetězcem začíná. Například máme-li řetězec `abcde` a zvolíme si podřetězec `bc`, pak se jedná o sufix `bcde`.

K čemu je toto pozorování dobré? Říká nám, že když známe nějakou informaci o sufixech daného řetězce, můžeme z ní často snadno odvodit obdobnou informaci o libovolném jeho podřetězci. Zatímco počet podřetězců závisí na délce daného řetězce kvadraticky, počet jeho sufixů je jen lineární, takže je dokážeme zpracovat efektivněji.

Na tomto pozorování je založená hlavní datová struktura, kterou si v tomto studijním textu ukážeme.

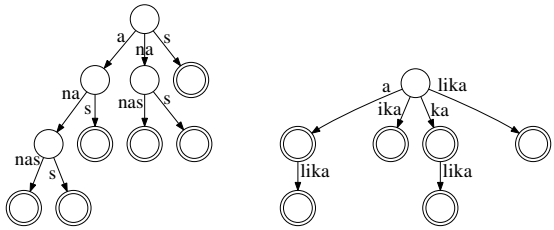
Sufixový strom

Sufixový strom odpovídající řetězci S je komprimovaný písmenkový strom obsahující množinu všech neprázdných sufixů řetězce S.

Například sufixový strom odpovídající řetězci `abcde` je vlastně komprimovaný písmenkový strom obsahující řetězce `abcde`, `bcde`, `cde`, `de` a `e`.

Kdybychom chtěli sufixový strom daného n-znakového řetězce sestrojit přímo podle naší definice, potřebovali bychom na to $\Theta(n^2)$ kroků: postupně po jednom bychom do něj vkládali všechny sufixy, jejichž součet délek je $n(n+1)/2$.

Všimněte si ale, že výsledný strom má nejvýše n listů (jeden pro každý sufix). Má tedy jenom $\mathcal{O}(n)$ vrcholů a také pouze $\mathcal{O}(n)$ hran. Zdá se proto, že bychom ho mohli sestrojit i v lepším čase než kvadratickém. Skutečně existují šikovné algoritmy, které k danému řetězci postaví jeho sufixový strom dokonce v čase $\Theta(n)$. Tyto algoritmy ovšem svou náročností přesahují rámec tohoto textu a nebudeme se jimi zde zabývat.



Vlevo sufixový strom pro řetězec `ananas`*, vpravo pro řetězec* `kalika`*.*

Sufixový strom se zarážkou

Sufixový strom pro řetězec `ananas` měl jednu pěknou vlastnost: každému sufixu odpovídal jeden z listů tohoto stromu. Sufixový strom pro řetězec `kalika` tuto vlastnost neměl, jelikož třeba sufix `a` je prefixem sufixu `alika`.

Tomu však můžeme snadno pomoci: namísto řetězce `kalika` sestrojíme sufixový strom pro řetězec `kalika#` (přičemž obecně `#` představuje libovolný symbol, který se v původním řetězci nevyskytuje). V novém sufixovém stromu už skutečně každý sufix odpovídá jinému listu, neboť po přidání „zarážky“ `#` na konec řetězce už zjevně nemůže být jeden sufix prefixem jiného.

Detaily reprezentace sufixového stromu v paměti

Než se pustíme do řešení soutěžních úloh, musíme se ještě domluvit na některých technických detailech.

- Sufixový strom je objekt. Obsahuje proměnnou `retezec`, v níž je uložen znakový řetězec, jehož sufixy jsou uloženy ve stromu. Dále obsahuje proměnnou `koren`, která představuje ukazatel na kořen samotného stromu.
- Každý vrchol stromu je objekt, který obsahuje tři proměnné:

- proměnnou `data`, do níž si můžete ukládat údaje libovolného typu (tento typ si můžete sami zvolit, jak potřebujete)
- proměnnou `deti`, což je pole indexované písmeny (prvním písmenem řetězce přiřazeného hraně). Každý prvek tohoto pole obsahuje ukazatel na příslušnou hranu. Pokud taková hrana neexistuje, je příslušný ukazatel nulový.
- proměnnou `konec`, v níž je uložena hodnota true nebo false podle toho, zda tu končí nějaký sufix

- Každá hrana stromu je také objekt. Obsahuje tři proměnné: číselné proměnné `od` a `do` a ukazatel na vrchol `kam`. Proměnná `kam` ukazuje na vrchol, do něhož hrana vede. Číselné proměnné říkají, že řetězec přiřazený této hraně je podřetězec původního řetězce (toho, který je uložen v proměnné `retezec` pro celý strom) tvořený znaky na pozicích `od` až `do-1` včetně.

  (Proč jsme použili proměnné `od` a `do` místo toho, abychom pro každou hranu přímo uložili její řetězec? Rozmyslete si, že kdybychom dotyčné řetězce zapisovali přímo, potřebovali bychom na uložení stromu v nejhorším možném případě kvadraticky mnoho paměti.)
- Ve svých řešeních můžete používat funkci `vytvor_strom(r)`, které předáte jako jediný parametr řetězec `r`, jehož sufixový strom chcete sestrojit. Funkce tento strom (v lineárním čase vzhledem k délce zadaného řetězce) postaví a vrátí ho jako návratovou hodnotu.

Rozšířený sufixový strom

Občas potřebujeme sufixový strom pro více než jeden řetězec. Máme například řetězce `A` a `B` a chceme sestrojit strom, který bude obsahovat sufixy řetězce `A` i sufixy řetězce `B`.

K tomu stačí šikovně využít funkci `vytvor_strom`. Na vstup jí předložíme řetězec `A#B#`, kde `#` („zarážka“) je nový znak nevyskytující se ani v `A`, ani v `B`. Ve stromu, který takto získáme, budeme ignorovat (nebo dokonce smažeme) všechno, co se nachází pod nějakým výskytem znaku `#`.

Například máme-li řetězce `macka` a `pes`, sestrojíme sufixový strom pro řetězec `macka#pes#`. V tomto stromu bude uložen třeba sufix `es#` (odpovídající sufixu `es` řetězce `pes`), ale také sufix `cka#pes#` (odpovídající sufixu `cka` řetězce `macka`).

Někdy je navíc užitečné použít navzájem různé zarážky. Když sestrojíme sufixový strom pro řetězec `macka$pes#`, můžeme pak rozlišit, zda sufix patří prvnímu nebo druhému řetězci podle toho, na kterou zarážku dříve narazíme při jeho čtení.

Příklad 1

Úloha: Na vstupu je zadán dlouhý řetězec `T`. Poté bude přicházet mnoho dalších řetězců. O každém z nich zjistěte, zda se v `T` nachází jako podřetězec.

Řešení: Sestrojíme si sufixový strom pro `T`. Následně pro každý řetězec `S` začneme v kořeni stromu a snažíme se sestupovat dolů cestou, která odpovídá řetězci `S`. Když se nám to podaří, řetězec `S` se v `T` nachází. Když někde cestou uvázneme a nemůžeme pokračovat dále, nastal opačný případ.

Každý řetězec takto zpracujeme v čase lineárním vzhledem k jeho délce.

```
def zjisti_zda_se_nachazi(strom, slovo):
    ''' Zjistí, zda se řetězec "slovo" nachází v řetězci T,
        jehož sufixový strom je "strom". '''

    kde = strom.koren  # začneme v kořeni stromu
    i = 0              # zpracujeme i-té písmeno řetězce "slovo"
    while i < len(slovo):
        # Zkontrolujeme, zda z aktuálního vrcholu vede hrana pro správné písmeno.
        if slovo[i] not in kde.deti: return False
        hrana = kde.deti[ slovo[i] ]

        # Pokud vede, zkontrolujeme, zda je celý text hrany správný.
        delka = min( hrana.do - hrana.od, len(slovo) - i )

        text_hrana = strom.retezec[ hrana.od : hrana.od + delka ]
        text_slovo = slovo[ i : i+delka ]
        if text_hrana != text_slovo: return False

        # Když text odpovídal, posuneme se o vrchol níže.
        i += delka
        kde = hrana.kam
    return True

T = input()
strom = vytvor_strom(T)

Q = int( input() )      # Počet otázek
for q in range(Q):
    slovo = input()     # Přečteme otázku
    print( zjisti_zda_se_nachazi( strom, slovo ) )
```

Příklad 2

Úloha: Na vstupu je zadán dlouhý řetězec `T`. Poté bude přicházet mnoho dalších řetězců. O každém z nich zjistěte, *kolikrát* se v `T` nachází jako podřetězec.

Řešení: Upravíme předchozí řešení. Až sestrojíme strom, rekurzívně ho projdeme a v každém vrcholu si spočítáme, kolik sufixů pod ním končí – tedy kolik vrcholů pod ním (včetně jeho samotného) má proměnnou `konec` nastavenu na true.

Rozmyslete si, že máme-li v našem sufixovém stromu vrchol r odpovídající řetězci `R`, potom každý konec sufixu v podstromu s kořenem r odpovídá jednomu výskytu řetězce `R` v původním textu. Namísto true/false tedy na zadanou otázku odpovíme naší spočítanou hodnotou.

V následujícím výpisu programu uvádíme jen ty části, v nichž se liší od předcházejícího.

```
def spocitej_konce(kde):
    kde.data = 0
    if kde.konec:
        kde.data = 1
    for x in kde.deti:
        kde.data += spocitej_konce( kde.deti[x].kam )
    return kde.data
```

```
def kolikrat_se_nachazi(strom,slovo):
    ''' zjistí, kolikrát se řetězec "slovo" nachází v řetězci T,
        jehož sufixovy strom je "strom" '''

    # ...
        if slovo[i] not in kde.deti: return 0
    # ...
        if text_hrana != text_slovo: return 0
    # ...
    return kde.data

T = input()
strom = vytvor_strom(T)
spocitej_konce( strom.koren ) # <--- před zpracováním otázek
                              # jednou spočítáme odpovědi

Q = int( input() )      # Počet otázek
for q in range(Q):
    slovo = input()     # Přečteme otázku
    print( kolikrat_se_nachazi( strom, slovo ) )
```