P-III-4 Nahoru a dolů

Stručný popis optimálního řešení: Určíme ty prvky posloupnosti, u nichž se nepožaduje, aby byly větší než nějaký jiný prvek. Ty všechny nastavíme na nulu a od nich následně odvodíme minimální možné hodnoty pro všechny ostatní prvky posloupnosti.

Lehké řešení za několik bodů

Nejprve popíšeme jednodušší řešení, za které jste mohli s minimem práce získat nějaké body. Vyhovující posloupnost celých čísel sestrojíme snadno: Začneme od libovolné výchozí hodnoty a potom po znacích zpracováváme daný řetězec. Když vidíme znak `=`, zopakujeme poslední hodnotu, když vidíme `<` nebo `>`, přidáme novou hodnotu, která je o 1 větší, resp. o 1 menší, než hodnota předcházející.

Například když začneme od 10, tak pro řetězec `<=>><` dostaneme posloupnost $(10, 11, 11, 10, 9, 10)$.

V prvních dvou testovacích vstupech stačilo začít třeba od hodnoty n. (Nebo například od hodnoty 500 000 000, která leží ve středu povoleného rozsahu.) To nám zaručilo, že všechny hodnoty, které použijeme v řešení, patří do povoleného rozsahu.

Na třetí testovací vstup stačilo uvedené řešení drobně vylepšit. Až získáme výše popsaným postupem *nějakou* posloupnost, můžeme nalézt její minimum a to odečíst od všech členů posloupnosti. V uvedeném příkladu bychom tak upravili naši posloupnost na $(1, 2, 2, 1, 0, 1)$.

Tímto způsobem dostaneme ve třetí sadě vždy platné řešení – až na dvě výjimky. Těmi jsou vstupy obsahující tisíc znaků `>`, resp. tisíc znaků `<`. Tyto dva vstupy nemají platné řešení, takže pro ně je třeba vypsat samé mínus jedničky.

Myšlenka vzorového řešení

Nyní budeme hledat *optimální* řešení, tedy řešení s nejmenším možným součtem členů.

Rádi bychom začali tím, že přestaneme uvažovat znak `=`. Na první pohled si to můžeme dovolit, neboť přece `=` vždy můžeme vyřešit tak, že najdeme řešení pro vstup bez znaků `=` a potom jenom zdvojíme ty hodnoty, které odpovídají výskytu `=`.

Toto bude skutečně fungovat – ale uvědomte si, že to není nic, co by automaticky muselo platit! Kdybychom použili jiné kritérium optimality, už by takové řešení fungovat nemuselo. Nic nám totiž (zatím) nezaručuje, že když vezmeme optimální řešení pro vstup bez `=`, dostaneme z něho výše popsaným postupem *optimální* řešení pro vstup s `=`. („Kdybych věděl, že tuto hodnotu mám sedmkrát zopakovat, zvolil bych ji menší a raději bych namísto toho zvolil větší jinou hodnotu, kterou budu mít v řešení jenom jednou!“ mohl by si stěžovat imaginární řešitel jiné, podobné úlohy.)

Zatím tedy uvažujme vstupy obsahující všechny tři druhy znaků. Časem si ukážeme, že v naší úloze opravdu můžeme znaky `=` vynechat. Pro některé vstupy je

optimální řešení zjevné. Například pro posloupnost `<<<` je to $(0, 1, 2, 3)$, pro `>>>` je to $(3, 2, 1, 0)$, pro `<=<=<` je to $(0, 1, 1, 2, 2, 3)$ a pro `<<>>` je to $(0, 1, 2, 1, 0)$.

Zamyslíme se, jak bude vypadat optimální řešení pro vstup `<<<<<>>`. Hledáme tedy posloupnost celých čísel $(x_0, x_1, \ldots, x_7)$ takovou, že platí

$$0 \leq x_0 < \ x_1 < x_2 < \ x_3 < x_4 < x_5 > \ x_6 > \ x_7 \geq 0.$$

Zleva si dokážeme postupně odvodit $x_0 \geq 0$, $x_1 \geq 1$, a tak dále až k $x_5 \geq 5$. Zprava si podobně odvodíme $x_7 \geq 0$, $x_6 \geq 1$ a $x_5 \geq 2$.

Pro x_5 jsme dostali dva různé odhady; silnější z nich je $x_5 \geq 5$. Pro každé jiné číslo jsme dostali jen jeden odhad. Nic proto nebrání tomu, abychom zvolili posloupnost, pro kterou bude *najednou ve všech odhadech* (samozřejmě kromě toho slabšího pro x_5) platit rovnost. Touto posloupností je v našem případě posloupnost $(0, 1, 2, 3, 4, 5, 1, 0)$. Výsledná posloupnost je nutně optimální – splňuje všechny požadované nerovnosti a o každém jejím členu umíme dokázat, že menší už být nemůže.

Výše popsaný postup dokážeme provést i pro zcela obecnou posloupnost znaků. Na papíru bychom takto už asi uměli k libovolné posloupnosti znaků ručně nalézt odpovídající optimální posloupnost hodnot a dokázat její optimálnost. My ale potřebujeme popsat exaktní (a pokud možno snadno implementovatelný) algoritmus, jak tuto posloupnost hodnot sestrojit.

V čem může nastat při implementaci problém? Například s těmi nešťastnými znaky rovná se. Když vidíme posloupnost znaků `><`, dostaneme posloupnost nerovností $x_{n-1} > \ x_n < \ x_{n+1}$, z níž je zjevné, že volba $x_n = 0$ nic nepokazí. Jak ale můžeme zabezpečit, že jakmile program uvidí posloupnost znaků `>===<`, přiřadí odpovídajícím proměnným nuly, ale zároveň neudělá totéž pro posloupnost `>===>`?

První možná implementace vzorového řešení

Kopcem nazveme posloupnost znaků, v níž se nejprve používají pouze znaky `<` a `=`, a následně se používají pouze znaky `>` a `=`. Jinými slovy, posloupnost znaků je kopcem právě tehdy, když se v ní nevyskytuje žádný znak `>` nalevo od nějakého znaku `<`. Kopci odpovídá posloupnost čísel, která nejprve neklesá a potom neroste.

Optimální řešení pro kopec sestrojíme snadno: až na speciální případy (viz níže) budou obě hodnoty na jeho koncích nuly, a od nich směrem dovnitř postupně zvyšujeme hodnoty. Zároveň s touto konstrukcí přímo dostáváme i důkaz optimálnosti.

Jestliže dostaneme libovolný obecný vstup, můžeme ho jednoduše rozložit na posloupnost kopců – dokud může následující znak patřit do aktuálního kopce, přidáme ho tam, a když už do něj patřit nemůže, začneme vytvářet nový kopec. Rozmyslete si, že nový kopec začneme vytvářet právě tehdy, když jsme právě přišli ke znaku `<` a aktuální kopec už obsahoval aspoň jeden znak `>`.

Příklad: vstup `>><=<<><<>=>==<<` bychom takto rozdělili na kopce `>>`, `<=<<>`, `<<>=>==` a `<<`.

Snadno se přesvědčíme, že optimální řešení pro celý vstup můžeme sestrojit tak, že najdeme optimální řešení pro každý kopec zvlášť a následně tato řešení spojíme

do jednoho. Stačí si uvědomit, že optimální řešení každého kopce bude začínat i končit nulou, takže při jejich spojování nenastane žádný konflikt. (Možnou výjimkou bude jen začátek prvního a konec posledního kopce, ty však s ničím nespojujeme.)

Pozorování o znacích „rovná se“

Výše popsané řešení nám kromě jiného ukazuje, že znaky = skutečně můžeme v první chvíli ignorovat, vyřešit vstup bez nich a následně je přidat zpět a zdvojit odpovídající prvky sestrojené posloupnosti. Odstranění/přidání znaků = totiž nezmění místa, kde náš algoritmus rozdělí vstup na jednotlivé kopce.

Druhá možná implementace vzorového řešení

Když už víme, že znaky = můžeme odstranit, dostaneme jednodušší řešení. Nejprve tedy odstraníme všechny =. V druhém kroku každé proměnné, která má být menší než všichni její sousedé, přiřadíme hodnotu 0. V třetím kroku jdeme od proměnných, které mají hodnotu 0, doleva i doprava „do kopce“ a přiřazujeme proměnným postupně rostoucí hodnoty. Na závěr přidáme zpět znaky = a zduplikujeme odpovídající prvky.

Příklad:

vstup bez =		>	<	<	<	<	>	>	>	<	<
2. krok		0							0		
3. krok, první 0	1	0	1	2	3	4			0		
3. krok, druhá 0	1	0	1	2	3	4	2	1	0	1	2

(Všimněte si, že při zpracování druhé nuly jsme „na vrchu kopce“ hodnotu 4 nepřepsali menší hodnotou 3.)

Třetí možná implementace vzorového řešení

Předchozí řešení lze implementovat ještě o něco pohodlněji. Začneme tím, že opět odstraníme znaky =. Následně použijeme algoritmus, který jsme popsali úplně na začátku, abychom sestrojili *nějakou* posloupnost, která splňuje zadané nerovnosti. Potom už jenom tuto posloupnost dvakrát projdeme – jednou zleva doprava a podruhé zprava doleva. Při každém průchodu se postupně díváme na každou hodnotu a snižujeme ji co nejvíce, jak je to jen možné bez porušení zadaných podmínek. Přidat zpět znaky = již umíme triviálně při výpisu řešení.

Rozmyslete si, že takto sestrojíme přesně totéž řešení jako předcházejícím postupem. (Kdy dostanou hodnotu 0 proměnné, které ji mají dostat? Co se stane ve zbytku průchodu zleva doprava? A co v následném průchodu zprava doleva?)

Všechna tři popsaná řešení se dají implementovat s optimální časovou složitostí $\Theta(n)$.

Existují také pomalejší řešení. Například ve třetím řešení je možné namísto průchodu zprava doleva provádět opakovaně průchody zleva doprava tak dlouho, až se při některém průchodu už žádná hodnota nezmění. Takové řešení má v nejhorším případě časovou složitost $\Theta(n^2)$ a mělo by být ohodnoceno 6 body.

Část bodů bylo možné získat i za řešení založená na jiných myšlenkách. Například lze využít zadanou hranici m a místo celých kopců rozdělit posloupnost

na „svahy nahoru“ a „svahy dolů“. Jestliže existuje platná posloupnost čísel, můžeme jednu takovou posloupnost sestrojit tak, že po svahu nahoru děláme kroky o 1, až v úplně posledním kroku směrem nahoru skočíme až na m. A naopak, cestou dolů vždy, když máme klesnout, klesneme o 1, až při úplně posledním klesnutí skočíme na 0. Pokud existuje způsob, jak udržet všechny hodnoty postupnosti v rozsahu 0 až m, takto se nám to určitě podaří. Toto řešení bylo ohodnoceno aspoň 5 body.

```
// Optimální řešení třetí z metod popsaných ve vzorovém řešení
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

void fix(int z, vector<int> &A, const string &nS) {
    // Co nejvíce snížíme hodnotu na indexu z.
    A[z] = 0;
    if (z>0             && nS[z-1]=='<') A[z] = max( A[z], A[z-1]+1 );
    if (z<int(A.size()) && nS[z]  =='>') A[z] = max( A[z], A[z+1]+1 );
}

int main() {
    int N, M; cin >> N >> M;
    string S; cin >> S;

    // Odstraníme znaky '=' a místo nich si o každé hodnotě zapamatujeme,
    // kolikrát ji chceme vypsat.
    // Zároveň s tím sestrojíme nějakou korektní posloupnost.
    vector<int> A(1,N);
    vector<int> pocet(1,1);
    string nS;
    for (char c:S) {
        if (c=='<') { A.push_back( A.back()+1 ); pocet.push_back(1); nS += c; }
        if (c=='=') ++pocet.back();
        if (c=='>') { A.push_back( A.back()-1 ); pocet.push_back(1); nS += c; }
    }

    // Projdeme posloupnost zleva doprava a následně zprava doleva.
    int Z = A.size();
    for (int z=0; z<N; ++z) fix(z,A,nS);
    for (int z=N-1; z>=0; --z) fix(z,A,nS);

    // Zjistíme, zda se posloupnost vejde do zadaného rozsahu a podle toho
    // buď vypíšeme posloupnost, nebo chybovou zprávu.
    bool fits = true;
    for (int i=0; i<Z; ++i) if (A[i]>M) fits = false;
    if (fits) {
        for (int i=0; i<Z; ++i) for (int j=0; j<pocet[i]; ++j)
            cout << (i+j?" ":"") << A[i];
    } else {
        for (int n=0; n<N; ++n) cout << (n?" -1":"-1");
    }
    cout << endl;
    return 0;
}
```

P-III-5 Vyvážené řetězce

Úlohy, v nichž hledáme souvislou posloupnost s nějakou konkrétní vlastností, jsou poměrně časté a také v tomto ročníku olympiády se takové objevily. Proto i k řešení této úlohy bylo možné přistoupit klasicky: Základním nástrojem bude spočítání vhodné informace pro všechny prefixy nějakého řetězce.

Pomalé řešení

První možností samozřejmě je postupně procházet všemi možnými podposloupnostmi a každou z nich ověřit zvlášť. Rozumný způsob procházení je takový, že si vždy určíme začátek a k němu zkoušíme všechny možné konce, přičemž postupně zvyšujeme délku vybrané podposloupnosti. Tímto způsobem se nový podřetězec liší od předcházejícího jen v jednom písmenu, takže většinu známých informací si můžeme ponechat a využít.

Nejjednodušší bude pamatovat si pro každé písmeno z naší abecedy, kolikrát se dosud vyskytlo ve zkoumané podposloupnosti. Při každém posunutí začátku si musíme pole s touto informací vynulovat. Při posunutí konce ale pouze připočítáme jedno písmeno, které jsme tím přidali. Pak už jen zkontrolujeme, zda máme stejný počet všech písmen. Je třeba uvědomit si, že ve výsledku se nemusí nacházet všechna písmena z abecedy, takže pokud počet výskytů některých písmen vyjde 0, je to v pořádku. Popsané řešení má (pro n-znakový řetězec a abecedu tvořenou k písmeny) časovou složitost $\mathcal{O}(kn^2)$.

Uvedené řešení můžeme ještě trochu zlepšit. Místo toho, abychom pro každý konec kontrolovali v čase $\Theta(k)$, kolikrát máme které písmeno, dokážeme tuto kontrolu provést v konstantním čase. Stejně jako v předchozím řešení si budeme pamatovat, kolikrát jsme v aktuálním úseku viděli které písmeno. Navíc si ale budeme pamatovat několik pomocných údajů: maximum m ze zapamatovaných počtů písmen, počet písmen, která mají právě m výskytů, a počet písmen, která mají aspoň jeden výskyt. Pomocí nich získáme řešení s časovou složitostí $\mathcal{O}(n^2)$.

Prefixy a dvoupísmenná abeceda

Předchozí řešení nám zaručuje zisk asi 4 bodů. Podívejme se tedy na další testovací sadu. Ta obsahuje abecedu složenou jen ze dvou písmen – a a b. Výrazně se však zvýšila délka řetězce. V podobných úlohách se často vyplatí zamyslet se, zda neumíme spočítat nějakou užitečnou informaci pro každý prefix. Prefixů je totiž pouze n (lineárně mnoho) a libovolný souvislý úsek získáme odečtením dvou prefixů.

Nejdelší úsek tvořený opakováním jednoho písmena najdeme triviálně. Co ale s úseky, které obsahují stejný počet a a b? Pamatovat si samostatně počet výskytů a a počet výskytů b v každém prefixu nám moc nepomůže. Máme-li prefix, který obsahuje znak a 4-krát a znak b obsahuje 5-krát (zkráceně $(4, 5)$), nevíme, jaký druhý prefix k němu přiřadit. Jestliže odečteme prefix $(3, 4)$, dostaneme úsek s jedním a a jedním b, což je dobré. Rovněž vyhovující je ale i prefix $(2, 3)$ nebo $(0, 1)$.

Co mají tyto prefixy společné? Přece rozdíl počtu a-ček a b-ček, které obsahují. Právě tento rozdíl bude pro nás podstatný. Je totiž velmi snadné zjistit, kdy se počet a a b rovná. Pokud totiž $a = b$, tak $a - b = 0$.

Řešení je teď už jednoduché. Postupně procházíme řetězec a udržujeme si proměnnou `rozdil`. Kdykoliv narazíme na a, zvýšíme ji o 1, při b ji naopak o 1 snížíme. Pro každý prefix si poznamenáme do vhodné datové struktury (`map`, set, případně i obyčejné pole, neboť hodnoty jsou z rozmezí od $-n$ do n) nalezený rozdíl a pozici, kde jsme ho dosáhli. Pro každou hodnotu proměnné `rozdil` si navíc pamatujeme jen její první výskyt. Pokaždé, když vypočítáme novou hodnotu proměnné `rozdil`, podíváme se, zda jsme již stejnou hodnotu měli někdy dříve. Pokud ano, víme, že úsek mezi jejím prvním a aktuálním výskytem tvoří vhodný vyvážený řetězec. Vypočítáme velikost a pozici tohoto úseku a porovnáme ji s dosud nejlepším řešením.

Tímto postupem vyřešíme i další vstupní sadu, která sice obsahuje třípísmennou abecedu, ale víme, že ve výsledném řetězci jsou nejvýše dvě z písmen abecedy. Existují jen tři možnosti, která dvě písmena to budou, takže můžeme všechny tři možnosti vyzkoušet (postupně počítáme $a - b$, $b - c$ a $a - c$). Uvědomte si ještě, že vždy, když najdete třetí písmeno (to, které jste si nezvolili, že bude ve výsledku), je třeba smazat obsah naší struktury a začít za ním počítat úplně od začátku.

Víceznakové abecedy

Posledním krokem ke vzorovému řešení této úlohy je zjistit, jak řešit úlohu s abecedou, která obsahuje více než 2 znaky. Začneme s abecedou $\{a, b, c\}$ a budeme předpokládat, že optimální řešení obsahuje všechny tři znaky. Jakým způsobem tedy přidáme informaci o písmenu c?

Když jsme používali jen dvě písmena, pamatovali jsme si rozdíl $a - b$ a věděli jsme, že když se tento rozdíl rovná 0, máme stejný počet písmen a a písmen b. Rozumné by tedy bylo pamatovat si také rozdíl $a - c$. Kdyby se obě tyto hodnoty rovnaly 0, znamenalo by to, že $a = b$ a $a = c$, takže také $b = c$. Přesně to požadujeme. Pro tři písmena si proto budeme pamatovat tyto dvě hodnoty jako dvojici čísel pro každý prefix. Jakmile najdeme stejnou dvojici pro jiný prefix, úsek mezi nimi je vyvážený, neboť $a - b = 0$ a $a - c = 0$.

To nás již vede k optimálnímu řešení. Vidíme, že máme-li třípísmennou abecedu, musíme vyzkoušet všechny možnosti, která písmena budou obsažena ve výsledném řetězci – existují 3 možnosti pro dvojice písmen a 1 možnost pro trojici.

Tento přístup je možné zobecnit i pro víceznakové abecedy. Mějme abecedu obsahující k různých znaků označených $\{x_1, x_2, \ldots x_k\}$. Musíme vyzkoušet každou možnost, které znaky budou ve výsledném řešení. Jakmile si zvolíme nějakou podmnožinu těchto znaků (takových podmnožin je 2^k), můžeme použít výše popsaný algoritmus, který se postupně dívá na všechny prefixy daného řetězce. Pokud zvolíme podmnožinu, která obsahuje ℓ prvků, musíme si pamatovat všechny dosažené $(\ell-1)$-tice čísel – rozdíly $(x_1 - x_2, x_1 - x_3 \ldots x_1 - x_\ell)$. Když se tato $(\ell-1)$-tice zopakuje, je odpovídající řetězec vyvážený, neboť počty všech písmen se v něm rovnají počtu výskytů písmena x_1.

Zbývá nám určit časovou složitost tohoto řešení. Naše prefixové řešení mělo složitost $\mathcal{O}(n \log n)$, neboť pro každý prefix jsme se dívali do `map`u (teď už je zapotřebí složitější datová struktura než pole, jelikož si ukládáme celé k-tice čísel). Navíc ale pracujeme s k-ticemi, takže práce s `map`em se k-krát zpomalí. Dostáváme tak časovou

složitost $\mathcal{O}(nk \log n)$. Toto řešení ovšem musíme spustit pro každou podmnožinu našich k písmen v abecedě. Celková časová složitost bude proto $\mathcal{O}(2^k kn \log n)$.

```
#include <cstdio>
#include <map>
#include <set>
#include <string>
#include <vector>
using namespace std;

#define For(i, n) for (int i=0; i<(n); i++)

int main() {
    char C[1000042];
    scanf("%s", C);
    string s=C;
    int n=s.length();
    set<char> Pom;
    For(i, n) Pom.insert(s[i]);
    vector<char> znaky(Pom.begin(), Pom.end());
    int k=znaky.size();
    map<vector<int>, int> M;
    vector<char> z;
    vector<int> nula;
    vector<int> stav;
    int zac=1, kon=0;
    for (int i=1; i<(1<<k); i++) {
        Pom.clear();
        z.clear();
        For(j, k)
            if (i&(1<<j)) z.push_back(znaky[j]);
        M.clear();
        For(j, z.size()-1) nula.push_back(0);
        M[nula]=-1;
        stav=nula;
        For(j, n) {
            int pom=-1;
            For(k1, z.size()) if (z[k1]==s[j]) pom=k1;
            if (pom==-1) {
                M.clear();
                M[nula]=j;
                stav=nula;
                continue;
            }
            if (pom==0) For(k1, z.size()-1) stav[k1]++;
            else stav[pom-1]--;
            if (M.find(stav)==M.end()) M[stav]=j;
            if (kon-zac<j-M[stav]-1) {zac=M[stav]+1; kon=j;}
            else if (kon-zac==j-M[stav]-1 && M[stav]+1<zac) {zac=M[stav]+1; kon=j;}
        }
    }
    printf("%d %d\n", zac, kon);
    return 0;
}
```

Lepší řešení

Úlohu lze řešit ještě lépe, k dosažení plného počtu bodů ale nebylo lepší řešení zapotřebí. V předchozím řešení se například můžeme zbavit faktoru k v časové složitosti tím, že si budeme pamatovat počty $a-b$, $b-c$, $c-d$, atd. a místo ukládání celé $(k-1)$-tice použijeme vhodné hešování.

Existují dokonce i řešení, jejichž časová složitost závisí na n přibližně lineárně (možná s faktorem $\log n$ navíc) a také na k závisí polynomiálně. Jednou z možností je začít tím, že si zvolíme počet x různých písmen, která má hledaný řetězec obsahovat. Následně ve vstupním řetězci najdeme všechny maximální úseky obsahující právě x různých písmen a každý z nich zpracujeme samostatně.

P-III-6 ACGT

Nejprve vyřešíme jednodušší problém: Pro zadané řetězce R, S chceme zjistit minimální počet změn potřebných na převedení R na S.

Tento problém vyřešíme dynamickým programováním. Postup se velmi podobá hledání nejdelší společné podposloupnosti. Budeme si vyplňovat matici A, kde hodnota $A[i][j]$ udává minimální počet změn potřebných na převedení prvních i znaků z řetězce R na prvních j znaků z řetězce S.

Jestliže $i=0$, počet změn je roven j; analogicky pro $j=0$. Jinak se podíváme na znaky $R[i-1]$ a $S[j-1]$. Jsou-li stejné, je zjevně optimální ponechat je. V takovém případě je optimálním řešením řešení pro $i-1$ znaků R a $j-1$ znaků S. Pokud se uvažované znaky liší, musíme to nějak napravit. Máme tři možnosti: můžeme smazat znak $R[i-1]$, můžeme vložit do R za pozici $i-1$ potřebný znak $S[j-1]$, nebo můžeme znak $R[i-1]$ změnit na znak $S[j-1]$. To vede k následujícím vztahům:

1. Jestliže $R[i-1]=S[j-1]$, potom $A[i][j]=A[i-1][j-1]$.
2. Jestliže $R[i-1]\neq S[j-1]$, potom $A[i][j]=\min(A[i-1][j]+1, A[i][j-1]+1, A[i-1][j-1]+1)$.

Výpočet hodnot pole A nám zabere čas přímo úměrný jeho velikosti, tedy $\mathcal{O}(|R|\cdot|S|)$.

To se dá ještě zlepšit, máme-li stanoven limit d na počet povolených změn. Všimněte si, že všechna políčka, kde $A[i][j]>d$, počítáme zbytečně, takže jejich výpočet můžeme vynechat. Lze ukázat, že potřebných políček bude $\mathcal{O}((|R|+|S|)\cdot d)$. Návod: pokud $|i-j|>d$, tak nutně $A[i][j]>d$.

Na právě popsaný algoritmus se můžeme podívat také z jiného úhlu. Vytvoříme si graf, jehož vrcholy jsou dvojice indexů. Přesněji, vrchol (i,j) představuje stav, v němž jsme převedli prvních i písmen řetězce R na prvních j písmen řetězce S.

V tomto grafu budou vrcholy spojeny orientovanou hranou délky 0, jestliže se mezi nimi dá přejít beze změny řetězců (případ 1 uvedený výše) nebo orientovanou hranou délky 1, jestliže to můžeme provést jednou změnou (případ 2 výše). V tomto grafu hledáme nejkratší cestu z vrcholu $(0,0)$ do vrcholu $(|R|,|S|)$. Jelikož hrany mají pouze ohodnocení 0 a 1, můžeme použít tzv. *0-1 prohledávání do šířky*, což je

v podstatě obyčejné prohledávání do šířky, ale s tím, že pokud vzdálenost do vrcholu v zlepšíme, přičemž jsme do něj přišli hranou délky 0, vložíme v nikoliv na konec, ale na začátek fronty.

Tento postup bude mít při dobré implementaci opět časovou složitost $\mathcal{O}((|R| + |S|) \cdot d)$.

Nyní se už můžeme pustit do řešení samotné úlohy. K získání 4 bodů stačí na vstupech, kde $n = 2$ a $m = 1$, správně implementovat výše uvedené postupy a ověřit tak, že jediné přípustné řešení skutečně má dostatečně málo chyb. Pátý bod dokážeme snadno získat ošetřením speciálního případu: jestliže $d = 0$, můžeme správnou posloupnost sestrojit hladově – stačí využít toho, že pokaždé, když potřebujeme přejít na nový fragment, umíme podle prvního písmena určit, na který.

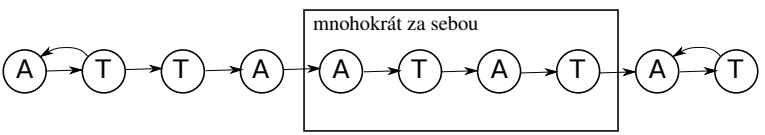
Přikročíme teď k řešení obecného případu. Potřebujeme rozšířit popsaný postup na netriviální graf – tedy situaci, kdy máme více než 2 fragmenty a mnoho různých dvojic fragmentů, které po sobě mohou následovat. Do stavu prohledávání nám při tomto rozšíření přibude fragment, v němž se právě nacházíme.

Přesněji, vrcholy našeho grafu budou teď trojice (i, f, j) přirozených čísel. Trojice (i, f, j) znamená, že momentálně jsme na fragmentu f, zpracovali jsme už j prvních písmen tohoto fragmentu, a společně s dříve zpracovanými fragmenty jsme vytvořili prvních i znaků řetězce R. Hrany v rámci fragmentu vypadají stejně jako ve výše popsaném řešení pro dva řetězce. Navíc budeme mít hrany délky 0 odpovídající tomu, že za jeden fragment přiložíme další, který za něj přiložit smíme.

V takovém grafu sestrojíme pomocí 0-1 prohledávání do šířky množinu všech jeho vrcholů, které jsou ve vzdálenosti nejvýše d od počátečního vrcholu $(0, u, 0)$. Nakonec se jenom podíváme, zda je mezi dosažitelnými vrcholy i cílový vrchol $(|R|, v, |F_v|)$. Pokud ano, sestrojíme cestu, kterou jsme ho dosáhli. Z ní snadno získáme použitou posloupnost fragmentů.

Na závěr už jenom odhadneme časovou složitost tohoto algoritmu. Označme r délku řetězce R, který sestrojujeme, a f součet délek fragmentů na vstupu. Snadný horní odhad je $\mathcal{O}(rf)$: pro každou pozici v řetězci R můžeme zároveň být na kterékoliv pozici kteréhokoliv fragmentu. Už tento horní odhad zajišťuje, že popsané řešení bude dostatečně efektivní pro většinu sad testovacích vstupů.

Na rozdíl od případu dvojice řetězců se pro tento algoritmus nedá dokázat lepší odhad. Představte si například fragmenty jako na obrázku (každý kroužek je jeden fragment):



Kdybychom hledali řetězec, který by byl složen z mnoha `AT` po sobě, začínali bychom fragmentem úplně vlevo, končili fragmentem úplně vpravo a povolili bychom 2 změny, tak bychom navštívili skoro každý stav.

V případě „slušných“ vstupů se ale ukazuje, že až tak mnoho stavů nenavštívíme, jelikož v případech víceznakových fragmentů potřebujeme, aby se celý fragment téměř shodoval s nějakou částí řetězce R, a toto nastane jen na málo místech.

```
#include <cstdio>
#include <vector>
#include <string>
#include <deque>
#include <unordered_map>
#include <algorithm>

using namespace std;
char buf[1000000];

struct Pos {            // vrchol grafu
  int tp;               // počet zpracovaných znaků z R
  int nod;
  int sp;               // počet zpracovaných znaků z aktuálního fragmentu
  Pos() {}
  Pos(int a, int b, int c) : tp(a), nod(b), sp(c) {}

  bool operator==(const Pos &b) const {
    return tp == b.tp && nod == b.nod && sp == b.sp;
  }
};

namespace std {

template<>
struct hash<Pos> {
  size_t operator()(const Pos& a) const {
    return hash<int>()(a.tp) ^ (hash<int>()(a.nod) << 3) ^ (hash<int>()(a.sp) << 5) ^
           (hash<int>()(a.tp) >> 2) ^ (hash<int>()(a.nod) >> 1) ^ hash<int>()(a.sp);
  }
};

}

int main() {
  int n, m; scanf("%d %d ", &n, &m);
  vector<string> nodes;
  for (int i = 0; i < n; i++) {
    scanf("%s", buf);
    nodes.push_back(string(buf));
  }

  vector<vector<int>> g(n);
  for (int i = 0; i < m; i++) {
    int a, b; scanf("%d %d ", &a, &b);
    a--; b--;
    g[a].push_back(b);
  }

  int d, start, end;
  scanf("%d %d %d ", &d, &start, &end);
  start--; end--;
  scanf("%s", buf);
  string target(buf);
```

```
deque<pair<Pos,int>> fr;
unordered_map<Pos, Pos> back;
back.reserve(target.size()*5*d);
fr.push_back(make_pair(Pos(0, start, 0), 0));
unordered_map<Pos, int> dists;
dists.reserve(target.size()*5*d);

while (!fr.empty()) {
  Pos x = fr.front().first;
  int dd = fr.front().second; fr.pop_front();
  if (dd > d) continue;
  if (dd > dists[x]) continue;
  if (x.tp == target.size() && x.nod == end && x.sp == nodes[x.nod].size()) {
    vector<int> path;
    Pos cp = x;
    path.push_back(cp.nod);
    while (back.count(cp)) {
      Pos prev = back[cp];
      if (prev.nod != cp.nod) {
        path.push_back(prev.nod);
      }
      cp = prev;
    }
    reverse(path.begin(), path.end());
    for (int i = 0; i < path.size(); i++) {
      printf("%d%c", path[i] + 1, i + 1 == path.size() ? '\n' : ' ');
    }
    return 0;
  }
  if (x.sp < nodes[x.nod].size()) {
    if (x.tp < target.size()) {
      if (nodes[x.nod][x.sp] == target[x.tp]) {
        Pos nx(x.tp+1, x.nod, x.sp+1);
        if (dists.count(nx) == 0 || dists[nx] > dd) {
          back[nx] = x;
          fr.push_front(make_pair(nx, dd));
          dists[nx] = dd;
        }
      } else {
        Pos nx(x.tp+1, x.nod, x.sp+1);
        if (dists.count(nx) == 0) {
          back[nx] = x;
          fr.push_back(make_pair(nx, dd+1));
          dists[nx] = dd+1;
        }
      }
    }
    Pos nx(x.tp, x.nod, x.sp+1);
    if (dists.count(nx) == 0) {
      back[nx] = x;
      fr.push_back(make_pair(nx, dd+1));
      dists[nx] = dd+1;
    }
  } else {
    for (int i = 0; i < g[x.nod].size(); i++) {
```

```
        int nnod = g[x.nod][i];
        Pos nx(x.tp, nnod, 0);
        if (dists.count(nx) == 0 || dists[nx] > dd) {
          back[nx] = x;
          fr.push_front(make_pair(nx, dd));
          dists[nx] = dd;
        }
      }
    }
    if (x.tp < target.size()) {
      Pos nx(x.tp+1, x.nod, x.sp);
      if (dists.count(nx) == 0) {
        back[nx] = x;
        fr.push_back(make_pair(nx, dd+1));
        dists[nx] = dd+1;
      }
    }
  }
  printf("-1\n");
  return 0;
}
```