P-I-1 Mezery

Velmi láká zkusit úlohu řešit hladově, tedy pokusit se na každý řádek výstupu přepsat tolik slov ze vstupu, kolik je jen možné. Toto řešení ale není optimální. Například uvažme vstup, skládající se z 50 jednopísmenných slov, následovaných slovem délky 29. Prvních 50 slov i s mezerami zabere alespoň 100 znaků, slovo délky 29 se tedy nevejde na první dva řádky a bude muset být až na třetím řádku. Hladové řešení by na první řádek dalo 30 jednopísmenných slov (šířka mezer 30/29) a na druhý 20 slov (šířka mezer 40/19). Chyba těchto dvou řádků je celkem 23,245. Na druhou stranu, kdybychom místo toho dali na první i druhý řádek 25 slov (šířka mezer 35/24), chyba by byla pouze 10,083.

Jistě by bylo možné zkoušet všechny možnosti zalomení textu. Například lze použít následující rekurzivní funkcí, která vrátí nejmenší možnou chybu textu skládajícího se z prvních n slov seznamu S s tím, že prozatím poslední řádek nepovažujeme za speciální (tj. započítáváme i jeho chybu a vyžadujeme, aby na něm byla alespoň dvě slova). V kódu $S(a, b)$ označuje podseznam S skládající se a-tého, $(a + 1)$-ního, ..., b-tého slova z S.

Procedura NejmenšíChyba(S, n)

1. Jestliže $n = 0$, vrátíme 0; jestliže $n = 1$, vrátíme $+\infty$.
2. *aktuální_nejlepší* $\leftarrow +\infty$
3. $p \leftarrow 2$
4. Dokud $p \leq n$, opakujeme následující:
5. Jestliže řádek obsahující slova $S(n - p + 1, n)$ má i s mezerami délku větší než 60, ukončíme cyklus.
6. *aktuální* $\leftarrow$ chyba řádku obsahujícího $S(n - p + 1, n) +$ NejmenšíChyba$(S, n - p)$
7. *aktuální_nejlepší* $\leftarrow \min($*aktuální_nejlepší*, *aktuální*$)$
8. $p \leftarrow p + 1$
9. Vrátíme *aktuální_nejlepší*.

Nakonec vyzkoušíme všechny možnosti pro poslední řádek, optimální chyba celého textu tedy bude minimum z NejmenšíChyba$(S, n - p)$ přes všechny volby p, kde $1 \leq p \leq n$ a řádek $S(n - p + 1, n)$ i s mezerami má délku nejvýše 60.

Toto řešení samozřejmě bude velmi pomalé. Pro text skládající se například z N slov délky 1 bude hloubka rekurze v každé větvi alespoň $N/30$ a každé volání procedury NejmenšíChyba na úrovni rekurze menší než $N/30$ vede k 29 rekurzivním voláním. Časová složitost je tudíž alespoň $\Omega(29^{N/30})$, tedy exponenciální.

Povšimněme si ale, že proceduru NejmenšíChyba voláme mnohokrát se stejným parametrem n. Nabízí se tedy si její hodnoty ukládat a nepočítat je znovu. Modifikovaná procedura vypadá takto:

Procedura NejmenšíChyba(S, n)

1. Jestliže $n = 0$, vrátíme 0; jestliže $n = 1$, vrátíme $+\infty$.
2. Pokud *předpočítáno*$[n] = 1$, vrátíme *hodnota*$[n]$.
3. *aktuální_nejlepší* $\leftarrow +\infty$
4. $p \leftarrow 2$
5. Dokud $p \leq n$, opakujeme následující:
6. Jestliže řádek obsahující slova $S(n - p + 1, n)$ má i s mezerami délku větší než 60, ukončíme cyklus.
7. *aktuální* $\leftarrow$ chyba řádku obsahujícího $S(n - p + 1, n) +$ NejmenšíChyba$(S, n - p)$
8. *aktuální_nejlepší* $\leftarrow \min($*aktuální_nejlepší*, *aktuální*$)$
9. $p \leftarrow p + 1$
10. *předpočítáno*$[n] \leftarrow 1$, *hodnota*$[i] \leftarrow$ *aktuální_nejlepší*
11. Vrátíme *aktuální_nejlepší*.

(Pole *předpočítáno* před zahájením výpočtu zinicializujeme na samé nuly.)

Tělo této procedury se provede nanejvýš N-krát a cyklus v ní obsažený má nejvýše 29 iterací (neboť 31 i jednopísmenných slov se na řádek nevejde), celková časová složitost tedy bude lineární $\mathcal{O}(N)$, a stejná bude i paměťová složitost.

V řešení není nutné používat rekurzi. Místo toho můžeme postupně vyhodnocovat NejmenšíChyba$(S, 0)$, NejmenšíChyba$(S, 1)$, ..., NejmenšíChyba(S, N) a tato volání pouze čtou předchozí výsledky z pole `hodnota`. Tuto variantu organizace výpočtu (které se také říká *dynamické programování*) využívá níže uvedený program. Z něj je také vidět, jak lze nejenom určit optimální hodnotu chyby, ale také vypsat příslušné zalomení textu.

```
#include <stdio.h>
#include <string.h>
#include <math.h>

#define MAX_SLOV 100000

typedef struct
{
  char *sl;
  int delka;
  double cena_zlomu;    /* Optimální cena řešení pro část textu před tímto
                           slovem, jestliže před ním provedeme zlom.  */
  int predchozi_zlom;   /* Předchozí zlom pro tuto optimální cenu.  */
  int zlomit;           /* Pomocná proměnná pro výpis řešení.  */
} slovo;

static slovo text[MAX_SLOV];
static int pocet_slov;

/* Vypíše řešení s optimální chybou takové, že poslední zlom se provede
   před slovem číslo POSLEDNI_ZLOM.  */

static void
vypis(int posledni_zlom)
```

```
{
  int z = posledni_zlom, i;
  const char *sep = "";

  while (z > 0)
    {
      text[z].zlomit = 1;
      z = text[z].predchozi_zlom;
    }

  for (i = 0; i < pocet_slov; i++)
    {
      if (text[i].zlomit)
        sep = "\n";
      printf("%s%s", sep, text[i].sl);
      sep = " ";
    }
  printf("\n");
}

/* Vrátí chybu za řádek obsahující celkem ZNAKU znaků, z nichž
   MEZER jsou mezery.  */

static double
chyba_radku(int znaku, int mezer)
{
  double delka_mezery = (60.0 - znaku + mezer) / mezer;
  double chyba_mezery = delka_mezery - 1;
  return mezer * chyba_mezery * chyba_mezery;
}

/* Určí nejmenší chybu za délky mezer v části textu skládající se
   z prvních PRED slov textu (poslední řádek se nechová speciálně).  */

static void
nejmensi_chyba(int pred)
{
  int znaku = text[pred - 1].delka;
  int mezer = 0;
  double nejlepsi_cena = INFINITY, akt_cena;
  int nejlepsi_zlom = -1, apred;

  for (apred = pred - 2; apred >= 0; apred--)
    {
      znaku += 1 + text[apred].delka;
      if (znaku > 60)
        break;
      mezer++;
      akt_cena = text[apred].cena_zlomu + chyba_radku(znaku, mezer);
      if (akt_cena < nejlepsi_cena)
        {
          nejlepsi_cena = akt_cena;
          nejlepsi_zlom = apred;
        }
    }

  text[pred].cena_zlomu = nejlepsi_cena;
  text[pred].predchozi_zlom = nejlepsi_zlom;
}
```

```
int main(void)
{
  char buf[100];
  int i, posledni, nejlepsi_zlom;
  double nejlepsi_cena;

  while (scanf("%s", buf) == 1)
    {
      text[pocet_slov].sl = strdup(buf);
      text[pocet_slov].delka = strlen(buf);
      pocet_slov++;
    }

  /* Určeme optimální chybu pro všechny prefixy textu.  */
  text[0].cena_zlomu = 0;
  text[1].cena_zlomu = INFINITY;
  for (i = 2; i < pocet_slov; i++)
    nejmensi_chyba(i);

  /* Vyzkoušejme všechy možnosti pro poslední řádek.  */
  nejlepsi_cena = text[pocet_slov - 1].cena_zlomu;
  nejlepsi_zlom = pocet_slov - 1;
  posledni = text[pocet_slov - 1].delka;
  for (i = pocet_slov - 2; i >= 0; i--)
    {
      posledni += 1 + text[i].delka;
      if (posledni > 60)
        break;
      if (text[i].cena_zlomu < nejlepsi_cena)
        {
          nejlepsi_cena = text[i].cena_zlomu;
          nejlepsi_zlom = i;
        }
    }

  vypis(nejlepsi_zlom);

  return 0;
}
```

P-I-2 Rušení stanic

Na vstupu jsme dostali souvislý neorientovaný graf s N vrcholy a M hranami. Nejprve si rozmysleme, že hledané pořadí vrcholů existuje pro každý možný vstup: k tomu stačí ukázat, že v každém souvislém grafu existuje vrchol, po jehož odebrání je zbylý graf stále souvislý.

Takový vrchol najdeme snadno tak, že v zadaném grafu uvážíme nejdelší možnou cestu (pokud je takových více, vybereme libovolnou) a vezmeme jeden její krajní vrchol u. Kdyby po jeho odebrání graf už nebyl souvislý, musel by existovat vrchol w, do kterého bychom se z druhého krajního vrcholu v vybrané cesty nedostali. V původním grafu bychom se do něj dostali jen přes vrchol u a složením vybrané cesty z v do u a cesty z u do w bychom dostali ještě delší cestu, což není možné, protože jsme vybrali nejdelší. Odebráním vrcholu u tedy souvislost neporušíme.

Již pomocí tohoto poznatku můžeme sestrojit algoritmus. Vyzkoušíme každý vrchol, zda jeho odstraněním neporušíme souvislost. Až najdeme vhodný vrchol, odstraníme jej a postup opakujeme. V jednom kroku tedy vyzkoušíme $\mathcal{O}(N)$ vrcholů a pro každý z nich v čase $O(N + M)$ otestujeme souvislost grafu třeba pomocí prohledávání do šířky. Kroků bude celkem N, takže se dostáváme na časovou složitost $\mathcal{O}(N^2 \cdot (N + M))$.

K rychlejšímu řešení se dostaneme drobnou úpravou argumentu z prvního odstavce. Abychom našli vhodný vrchol k odstranění, nemusíme volit nejdelší cestu v celém grafu, stačí zvolit libovolný vrchol a z něj prohledat graf a najít od něj nejvzdálenější vrchol. Tento vrchol můžeme bez obav odstranit, protože kdyby porušil souvislost, byl by nějaký vrchol ještě dál od startovního než on, což nelze. Takovýto vrchol snadno najdeme pomocí prohledání grafu do šířky, bude to například ten, který navštívíme jako úplně poslední. Odstraňujeme N vrcholů, tento postup tedy provedeme N-krát, dostáváme tedy algoritmus běžící v čase $\mathcal{O}(N \cdot (N + M))$.

Optimálně rychlý algoritmus dostaneme, pokud si uvědomíme, že prohledávání do šířky provedené v předchozím řešení není třeba provádět za každý odstraňovaný vrchol, nýbrž jej stačí provést pouze jednou a zachovat si vrcholy grafu v pořadí, v jakém vstoupily do fronty. Tím máme všechny vrcholy uspořádáné podle vzdálenosti od jednoho konkrétního vrcholu a tudíž je lze díky pozorování v předešlém odstavci jednoduše vypsat od nejvzdálenějšího vrcholu. Tím jsme snížili časovou složitost na $\mathcal{O}(N + M)$.

Optimální algoritmus snadno dostaneme i pomocí prohledávání do hloubky. Stačí si uvědomit, že každý souvislý graf má kostru, ta je stromem a tudíž má list, jehož odstraněním není porušená souvislost. Stačí tedy najít nějakou kostru grafu a z té potom postupně odebírat listy. Jednu konkrétní kostru dostaneme přímo prohledáním grafu do hloubky. Výsledné pořadí vrcholů pak můžeme získat rovnou při průchodu, pokud daný vrchol vypíšeme, když se z něj rekurze vrací, neboť v tom okamžiku už všechny vrcholy ve stromě pod ním jsou vypsány a on je tím pádem listem.

```
/*** Řešení prohledáváním do hloubky ***/

#include <cstdio>
#include <vector>
#include <stack>

using namespace std;

#define MAXN 1000000
#define MAXM 10000000

/* Počet vrcholů a hran.  */
int N, M;

/* Seznam hran grafu.  */
struct edge { int u, v; } G[MAXM];

/* Sousedi vrcholu x jsou E[V[x]], E[V[x] + 1], ..., E[V[x+1] - 1] */
int V[MAXN+1];
int E[2*MAXM];

/* Pomocné proměnné pro prohledávání do hloubky.  */
int used[MAXN];
int neigh[MAXN];
int camefrom[MAXN];

int main()
{
        /* Načteme graf a spočítáme si seznamy sousedů do pole E.  */
        scanf("%d%d", &N, &M);
        for (int i = 0; i < M; i++) {
                scanf("%d%d", &G[i].u, &G[i].v);
                V[--G[i].u]++;
                V[--G[i].v]++;
        }
        for (int i = 0; i < N; i++)
                V[i+1] += V[i];
        for (int i = 0; i < M; i++) {
                E[--V[G[i].u]] = G[i].v;
                E[--V[G[i].v]] = G[i].u;
        }

        for (int i = 0; i < N; i++) {
                neigh[i] = V[i];
        }

        /* Prohledávání do hloubky.  */
        camefrom[0] = -1;
        int curvert = 0;

        while (curvert != -1) {
                used[curvert]++;
                if (neigh[curvert] < V[curvert+1]) {
                        if (used[E[neigh[curvert]]] == 0) {
                                camefrom[E[neigh[curvert]]] = curvert;
                                neigh[curvert]++;
                                curvert = E[neigh[curvert]-1];
                        } else {
                                neigh[curvert]++;
                        }
```

```
                } else {
                        printf("%d%s", curvert + 1, (curvert == 0) ? "\n" : " ");
                        curvert = camefrom[curvert];
                }
        }
        return 0;
}

/*** Řešení prohledáváním do šířky ***/

#include <cstdio>
#include <vector>

using namespace std;

#define MAXN 1000000
#define MAXM 10000000

/* Počet vrcholů a hran.  */
int N, M;

/* Sousedi vrcholů v grafu.  */
vector<int> Graph[MAXN];

/* Pomocné proměnné pro prohledávání do šířky.  */
int Queue[MAXN];
int Qidx;
bool inQueue[MAXN];

int main()
{
        /* Načtení vstupu.  */
        scanf("%d%d", &N, &M);
        for (int i = 0; i < M; i++) {
                int u, v;
                scanf("%d%d", &u, &v);
                u--; v--;
                Graph[u].push_back(v);
                Graph[v].push_back(u);
        }

        /* Prohledávání do šířky.  */
        Queue[Qidx++] = 0;
        inQueue[0] = true;

        for (int i = 0; i < Qidx; i++) {
                int deg = (int) Graph[Queue[i]].size();
                for (int j = 0; j < deg; j++) {
                        if (!inQueue[Graph[Queue[i]][j]]) {
                                inQueue[Graph[Queue[i]][j]] = true;
                                Queue[Qidx++] = Graph[Queue[i]][j];
                        }
                }
        }

        for (int i = Qidx - 1; i >= 0; i--) {
                printf("%d%s", Queue[i] + 1, (i > 0) ? " " : "\n");
        }
        return 0;
}
```

P-I-3 Ztracený paket

Omezení na paměť nám znemožňují zpracovávat naráz větší množství prvků ze vstupu, což vylučuje například možnost si pro každé číslo pamatovat, zda jsme ho již viděli. Kdyby byla čísla na vstupu setříděná dle velikosti, stačilo by je projít jednou a hledat „díru“, tj. dvě následující čísla na vstupu, která se liší o 2. Jedním přijatelným řešením (které může získat až 7 bodů) je tedy si vstupní soubor setřídit s použitím pomocných souborů nějakou metodou nevyžadující udržování tříděné posloupnosti v paměti, například Mergesortem.

Takové řešení ale vyžaduje číst a zapisovat soubory velikosti řádově n celkem $\mathcal{O}(\log n)$-krát. Existuje i jednodušší a rychlejší řešení s lineární časovou složitostí založené na vyhledávání chybějícího čísla metodou půlení intervalů, které ale stále vyžaduje použití pomocných souborů.

Mnohem efektivněji lze úlohu vyřešit s použitím jednoduchého pozorování: součet posloupnosti čísel od 1 do n v libovolném pořadí je vždy stejný (rovný $N = n(n+1)/2$, což ale k řešení úlohy ani nemusíme vědět, jelikož tento součet můžeme prostě určit přímo sečtením všech těchto čísel v programu). Bude-li v této posloupnost jedno číslo x chybět, bude se součet od N lišit přesně o x. Chybějící číslo můžeme určit jako

$$(\text{součet čísel od 1 do } n) - (\text{součet čísel ve vstupním souboru}).$$

Při vyjadřování těchto součtů je třeba být trochu opatrný kvůli omezenému rozsahu celočíselných typů. Základní celočíselný typ je často 32-bitový, takže se do něj vejde číslo $n \leq 1\,000\,000\,000$, ale už ne nutně součet čísel od 1 do n. Toto můžeme vyřešit prováděním výpočtů v 64-bitovém typu, nebo použitím jiné vhodné operace místo sčítání (například operace XOR), pro niž problém s přetečením nenastává. Případně si můžeme všimnout, že algoritmus může fungovat i v 32-bitovém typu, je-li pro něj aritmetika definována jakožto počítání se zbytky modulo 2^{32}, tedy jestliže $a + b$ je větší nebo rovno 2^{32}, je výsledkem sčítání v příslušném typu číslo $(a + b) \bmod 2^{32}$ (takto je například definováno chování sčítání pro neznaménkové typy v programovacím jazyce C). V tom případě výsledek výpočtu bude roven

$$((\text{součet čísel od 1 do } n) - (\text{součet čísel na vstupu})) \bmod 2^{32} = x \bmod 2^{32}.$$

To je ovšem rovno x, jelikož $0 \leq x \leq n < 2^{32}$.

```
#include <stdio.h>

int main(void)
{
        unsigned int i, n, s = 0, a;

        scanf("%u", &n);
        for (i = 1; i <= n - 1; i++) {
                scanf("%u", &a);
                s += i - a;
        }

        printf("%d\n", s + n);
        return 0;
}
```

P-I-4 Magická síť

Úkol 1: Uvažme síť $\text{NAE}(x, y, a)$, $\text{NAE}(a, t, o)$, $\text{ONE}(o)$, $\text{NAE}(t, t, z)$ se vstupními proměnnými x, y a z. Jestliže $x = 1$ nebo $y = 1$, pak můžeme položit $a = 0$, $o = 1$ a $t = 1 - z$, čímž zaručíme splnění všech omezení nezávisle na ohodnocení z. Jestliže $x = y = 0$, pak musíme položit $a = o = 1$, díky čemuž omezení $\text{NAE}(a, t, o)$ vynucuje $t = 0$ a omezení $\text{NAE}(t, t, z)$ vynucuje $z = 1$. Tato síť tedy přijímá právě ta ohodnocení, kde x, y nebo z mají hodnotu 1, a proto simuluje omezení $\text{OR}(x, y, z)$.

Úkol 2: Síť $\text{NAE}(a, x, x)$ přijímá právě ta ohodnocení, kde $a \neq x$. Proto síť $\text{NAE}(a, x, x)$, $\text{NAE}(b, x, x)$ vynucuje $a \neq x$ a $b \neq x$, a tedy $a = b$. Existuje i řešení, v němž se v rámci omezení neopakují proměnné. Síť $\text{NAE}(a, x, y)$, $\text{NAE}(a, x, z)$, $\text{NAE}(a, y, z)$ vynucuje, že nejvýše jedna z proměnných x, y a z má stejnou hodnotu jako a. Síť $\text{NAE}(a, x, y)$, $\text{NAE}(a, x, z)$, $\text{NAE}(a, y, z)$, $\text{NAE}(b, x, y)$, $\text{NAE}(b, x, z)$, $\text{NAE}(b, y, z)$ toto zaručuje jak pro a, tak pro b, a proto není možné, aby ve splňujícím ohodnocení měli a a b různé hodnoty.

Úkol 3: Jestliže hodnoty a, b a c splňují omezení NAE, tj. nejsou si všechny rovny, pak také hodnoty $1 - a$, $1 - b$ a $1 - c$ splňují omezení NAE. Uvažme libovolnou síť s proměnnými $x_1, \ldots, x_n$, používající pouze omezení typu NAE. Jestliže ohodnocení $x_1 = a_1, \ldots, x_n = a_n$ splňuje všechna omezení této sítě, pak je zřejmě splňuje i opačné ohodnocení $x_1 = 1 - a_1, \ldots, x_n = 1 - a_n$. Kdyby tato síť simulovala $\text{OR}(x_1, x_2, x_3)$, pak by existovalo ohodnocení s hodnotami $x_1 = x_2 = x_3 = 1$ přijímané touto sítí. Nicméně pak i opačné ohodnocení $x_1 = x_2 = x_3 = 0$ by bylo přijímané sítí, ale je odmítané omezením $\text{OR}(x_1, x_2, x_3)$. To je spor, tedy žádná síť obsahující pouze omezení typu NAE nemůže simulovat OR.