Rubrika: Nešto izmedju [rešenja problema]

časopis Tangenta, broj 02, godina 2010 / 2011

Problem: NI70 Štreber žurka

Zadatak se može uraditi na više načina. Ovde ćemo izneti jedan od efikasnijih. Pre svega vidimo da ukoliko dečak A može plesati sa devojkama B i C, tada su i imena devojaka B i C takodje anagrami. Drugim rečima relacija anagram je relacija ekvivalencija. Sam poredak slova u imenima osoba nam nije bitan, tako da svaku osobu možemo pamtiti kao strukturu od dva elementa

- *duzina* - dužina imena osobe
- niz *slova* dužine 26 - frekvenca, broj pojavljivanja, svakog slova u imenu

Momak i devojka mogu plesati zajedno akko su im imena iste dužine i nizovi frekvenci se poklapaju. Prvi uslov nije neophodan ako drugi važi, pošto ga povlači, ali nam može ubrzati ispitivanje. Sada možemo definisati klasu kao skup svih osoba čija su imena anagrami. Drugim rečima definisaćemo klasu kao:

- *duzina* - dužina imena osobe iz klase
- niz *slova* dužine 26 - frekvenca, broj pojavljivanja, svakog slova u imenima osoba iz klase
- $brojMomaka$ - broj momaka u klasi
- $brojDevojaka$ - broj devojaka u klasi

Redom ćemo ubacivati osobe u klase. Ukoliko postoji klasa čije je ime anagram osobe, onda je ubacujemo u nju i povećavamo broj momaka ili devojaka u zavisnosti pola osobe. Inače, kreiramo novu klasu čiji će, za sada, jedini predstavnik biti trenutna osoba.

Svaka osoba iz klase može plesati samo sa osobama suprotnog pola iz iste klase. Zbog ovog uslova, klase možemo posmatrati odvojeno. Maksimalni broj parova u svakoj klasi je jednak minimumu od broja momaka i devojaka. Krajnji rezultat predstavlja sumu broja parova po svim klasama.

```
#include<stdio.h>
#include<string.h>
#define MAX_N 105
#define MAX_M 105

// Struktura osobe
struct Osoba
{
    int duzina; // duzina imena
    int slova [26]; // frekvenca svakog slova u imenu
} osoba;

// Klase momaka i devojka cija imena su anagrami
struct Klasa
{
    int slova [26];  // frekvenca svakog slova u imenu klase
    int duzina;  // duzina imena klase
    int brojMomak;  // broj momaka u klasi
    int brojDevojka; // broj devojaka u klasi
} klasa;

int n, m, sol, brojKlasa;
Osoba momci [MAX_N], devojke [MAX_M];
Klasa klase [MAX_N + MAX_M];

    // Funkacija za unos podataka
```

```
void input()
{
    char s [21];
    scanf ("%d %d", &n, &m);
    // Unos momaka
    for (int i = 0; i < n; i++)
    {
        scanf ("%s", &s);
        momci [i].duzina = strlen(s);
        for (int j = 0; j < 26; j++)
            momci [i].slova [j] = 0;
        for (int j = 0; j < momci [i].duzina; j++)
            momci [i].slova[s [j] - 'a']++;
    }

    // Unos devojaka
    for (int i = 0; i < m; i++)
    {
        scanf ("%s", &s);
        devojke [i].duzina = strlen(s);
        for (int j = 0; j < 26; j++)
            devojke [i].slova [j] = 0;
        for (int j = 0; j < devojke [i].duzina; j++)
            devojke [i].slova[s [j] - 'a']++;
    }
}

// Funkacija minimuma dva broja
int min(int a, int b)
{
    if (a < b) return a;
    return b;
}

// Da li data osoba pripada klasi
bool Anagrami(Osoba osoba, Klasa klasa)
{
    if (osoba.duzina != klasa.duzina)
        return false;
    for (int i = 0; i < 26; i++)
        if (osoba.slova [i] != klasa.slova [i])
            return false;
    return true;
}

// Dodavanje nove osobe u klase
void Dodaj(Osoba osoba, bool musko)
{
    int index = 0;
    // Da li osoba pripada nekoj od postojecih klasa
    while (index < brojKlasa)
    {
        if (Anagrami (osoba, klase [index]))
            break;
        index++;
    }

    if (index == brojKlasa)
    {
        // Osoba definise novu klasu
        for (int i = 0; i < 26; i++)
            klase [index].slova [i] = osoba.slova [i];
        klase [index].duzina = osoba.duzina;
        klase [index].brojMomak = klase [index].brojDevojka = 0;
        brojKlasa++;
    }

    if (musko)
        klase [index].brojMomak++;
    else
        klase [index].brojDevojka++;
}

// Metoda koja resava problem
```

```
    void solve()
    {
        // Inicijalizacija klasa
        brojKlasa = 0;
        for (int i = 0; i < n; i++)
            Dodaj(momci [i], true);
        for (int i = 0; i < n; i++)
            Dodaj(devojke [i], false);
        // Broj parova u klasi je minimum od broja momaka i devojaka
        sol = 0;
        for (int i = 0; i < brojKlasa; i++)
            sol = sol + min(klase [i].brojMomak, klase [i].brojDevojka);
    }

    // Glavna metoda
    int main()
    {
        input();
        solve();
        printf ("%d\n", sol);

        return 0;
    }
```

Problem: NI71 Bubamara

Ovaj prelepi problem sa Županjskog takmičenja, Hrvatska 2007, može na jako konstruktivan način da ilustruje šta je to složenost algoritma. Naivan pristup ovom problemu je sledeći: za svaku moguću visinu, kojih ima h, prolazimo redom po svim preprekama i brojimo one na koje bubamara nailazi. Kako za svaku visinu obilazimo sve prepreke, kojih ima ukupnu n, dobijamo da je složenost ovog algoritma $O(h \cdot n)$.

Medjutim, ovaj problem možemo rešiti na efikasniji način. Ideja je da iskoristimo to što se po visinama krećemo redom. Naime, možemo primetiti sledeće: ukoliko na visini k bubamara ne udara u stalagmit A, tada ona neće udarati u njega ni za bilo koju veću visinu. Analogno, ukoliko bubamara nailazi na stalaktit B na datoj visina k, tada će nailaziti na njega za svaku visinu veću od k. Ova svojstva možemo iskoristiti ukoliko na početku sortiramo stalaktite i stalagmite. Pretpostavimo da smo stalagmite ubacili u niz a, a stalaktite u niz b, koje smo zatim sortirali u neopadajućem poretku. Na početku postavićemo dva pokazivača, i i j tako da i pokazuje na početak niza a dok j pokazuje na kraj niza b. Oni će u svakom trenutku pokazivati na element počev od koga bubamara udara u prepreke: za stalagmite ulevo (u sve veće) a za stalagmite udesno (u sve manje). Kada prelazimo na sledeću visinu, jednostavno pomeramo ove pokazivače sve dok ne naidjemo na one prepreke na koje bubamara udara. Kako se na ovaj način svaka prepreka obidje najviše jednom (pokazivači ne mogu ispasti iz opsega niza), složenost ovog algoritma nakon sortiranja je linearna. Ukupna složenost algoritma je $O(h + n \log n + m \log m)$.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_N 1005

int n, m, a [MAX_N], b [MAX_N], h, sol, num;

    void input()
    {
        scanf ("%d %d", &num, &h);
        n = m = 0;
        for (int i = 0; i < num; i++)
        {
            if (i % 2 == 0)
            {
                scanf ("%d", &a [n]);
                n++;
```

```
        }
        else
        {
            scanf ("%d", &b [m]);
            m++;
        }
    }
}

int compare (const void * a, const void * b)
{
    return ( *(int*)a - *(int*)b );
}

void solve()
{
    sol = 0;
    qsort (a, n, sizeof(int), compare);
    qsort (b, m, sizeof(int), compare);

    int min = n + m + 1;
    int i = 0, j = m - 1;
    for (int k = 1; k < h; k++)
    {
        while ((i < n) && (a [i] <= k))
            i++;
        while ((j >= 0) && (h - b [j] < k))
            j--;

        if ((n - i) + (m - 1 - j) == min)
            sol++;
        if ((n - i) + (m - 1 - j) <  min)
        {
            min = n - i + m - 1 - j;
            sol = 1;
        }
    }
}

int main()
{
    input();
    solve();
    printf("%d\n", sol);

    return 0;
}
```

Problem: NI72 Podstringovi

Problem najdužeg zajedničkog podstringa (odnosno podniza) je već razmatran u tekstu o Dinamičkom programiranju. Medjutim ovde se ne traži dužina NZP već svi NZP-ovi. Označimo početne stringove sa a i b, a njihove dužine sa n odnosno m. Kod samog problema NZP-a konstruisali smo matricu d dimenzije $n \times m$ i definisali je kao

$$d[i][j] = \text{ dužina najdužeg zajedničkog podstring za podstringove } a^i \text{ i } b^j$$

Rekurentna relacija kojom smo računali elemente matrice d glasila je:

$$d[x][y] = \begin{cases} 0, & \text{ako je } x \cdot y = 0 \\ d[x-1][y-1] + 1, & \text{ako je } a_x = b_y \\ \max\{d[x-1][y], d[x][y-1]\}, & \text{inače} \end{cases}$$

Nulta kolona i vrsta su dodate samo radi lakše implementacije algoritma.

Dužina najd̆užeg zajedničkog podstringa je $d[n][m]$. Nakon inicijalizacije matrice d, da bi rekonstruisali NZP-e, moramo se vraćati 'unazad' po matrici d po putevima koji vode do maksimalnih NZP-a. Ovo se najednostavnije može implementirati rekurzivno.

Algoritam: Pseudo kod rekurzivnog algoritma *traceback* za nalaženje svih NRP-a

Input: Matrica d definisana gore; prirodni brojevi x i y trenutna pozicija u matrici; string *currentSequence*

Output: svi NZP-ovi polaznih stringova a i b

```
1  if x = 0 or y = 0 then
2  |  dodaj currentSequence u skup rešenja;
3  |  return
4  end
5  if a[x] = b[y] then
6  |  traceback(d, x − 1, y − 1, currentSequence + a[x]);
7  |  return
8  end
9  if d[x − 1][y] = d[x][y] then
10 |  traceback(d, x − 1, y, currentSequence);
11 end
12 if d[x][y − 1] = d[x][y] then
13 |  traceback(d, x, y − 1, currentSequence);
14 end
```

Ukoliko malo bolje pogledamo ugore opisani pseudo kod rekurzivnije funkcije *traceback*, možemo primetiti da u slučaju kada je $d[x-1][y] = d[x][y-1]$ imamo dva moguća puta. Bez obzira što se u samoj postavci zadatka garatuje da broj različitih NZP-a neće biti veći od 1000, to ništa ne govori o broju načina na koji ih možemo dobiti. Za stringove *aaaaaaabccccccccd* i *abbbbbbbcddddddd* postoji čak 1.778.966 načina da se rekonstruiše jedinstveni NZP *abcd*. Ovaj problem možemo rešiti na dva načina: menjanjem definicije matrice d ili rekurzivne funkcije za rekonstrukciju NZP-a. Razmotrićemo oba pristupa.

Jedan od način da izbegnemo konstrukciju istih podstringova jeste da pri samoj evaluaciji maksimalne dužine vršimo i njihovu konstrukciju. Kako bi ovo izveli potrebno je da definišemo, pored matrice d, još jednu matricu *set* istih dimenzija. Matricu definišemo kao:

$$set[x][y] = \text{ skup svih mogućih najdužih podstringova stringova } a^x \text{ i } b^y$$

Kako inicijalizujemo elemente matrice *set*? Kao što znamo, pri računanju vrednosti elementa matrice d, element $d[x][y]$ se inicijalizuje preko jedne od vrednosti $d[x-1][y-1]$, $d[x-1][y]$ ili $d[x][y-1]$. Slično će i $set[x][y]$ naslediti jedan od prethodnih *set*-ova uz malu modifikaciju. Drugim rečima imamo četiri slučajeva pri inicijalizaciji skupa $set[x][y]$:

- $a[x] = b[y]$: tada imamo da se element iz $set[x][y]$ dobija kada se na neki string iz $set[x-1][y-1]$ doda karakter $a[x]$;
- $a[x] \neq b[y]$ i $d[x-1][y] > d[x][y-1]$: tada imamo da su skupovi $set[x][y]$ i $set[x-1][y]$ jednaki;
- $a[x] \neq b[y]$ i $d[x-1][y] < d[x][y-1]$: tada imamo da su skupovi $set[x][y]$ i $set[x][y-1]$ jednaki;
- $a[x] \neq b[y]$ i $d[x-1][y] = d[x][y-1]$: u ovom slučaju string pripada skupu $set[x][y]$ akko pripada barem jednom od skupova $set[x][y-1]$ i $set[x-1][y]$. Dakle, ovde imamo da je $set[x][y]$ jednak uniji skupova $set[x][y-1]$ i $set[x-1][y]$;

Ostaje još problem memorije. Kako ima najviše 1000 najdužih podstringova, a dužine stringova nisu veće od 100, u najgorem slučaju, zavisno od same implementacije matrice *set*, ovo će predstavljati problem. Medjutim, kako vidimo da su nam potrebna samo dva reda matrica d i *set*, tako da pri implementaciji ove ideje treba pamtiti samo poslednja dva reda opisanih matrica.

Pored ovog rešenja, postoji dosta brže rešenje. Ideja je da se koristi modifikovana verzija rekurzivne funkcije *traceback* opisane na početku. Ideja je sasvim prirodna: konstruisaćemo samo one

najduže podstringove x za koje je karakter $x[i]$ prvo pojavljivanje tog karaktera nakon pozicije karaktera $x[i-1]$. Sve ostale mogućnosti neće voditi do novih NRP-a tako da ih možemo zanemariti prilikom pretrage. Implementaciju ove ideje ostavljamo čitaocu.

Programerski trikovi

Ovaj problem je autor prikupio sa nekih intervijua programerskih firmi. Zadatak se može uraditi na više način.

- Prva verzija problema je dosta lakša. Na početku možemo sa p označiti proizvod svih elemenata niza a. Kako možemo koristiti operaciju deljenja, imamo da je $b[i] = p/a[i]$. Ovo je svakako linearni algoritam.
- Drugi podporblem je malo komplikovaniji. Definisaćemo dva niza $prefix$ i $sufix$ kao:

$$prefix[i] = a[1] \cdot a[2] \cdot \ldots \cdot a[i]$$

$$sufix[i] = a[i] \cdot a[i+1] \cdot \ldots \cdot a[n]$$

  Ove nizove možemo inicijalizovati linearno. Sada elemente niza b možemo računati kao:

$$b[i] = prefix[i-1] \cdot sufix[i+1]$$