Podatkovne strukture in algoritmi 1: 1. izpit

5. 2. 2019

Čas pisanja je 120 minut. Možno je doseči 80 točk. Veliko uspeha!

1. naloga (20 točk)

a) (10) Izvedi naslednjih 10 operacij na prazni max-kopici:

`push(1), push(3), push(8), push(5), push(4), pop(), push(7), pop(), pop(), push(2)`

Po vsaki operaciji nariši stanje kopice in nekam zapisuj vrstni red odstranjenih elementov. Stanje kopice po izvedbi operacije `push(4)` tudi predstavi s seznamom.

Rešitev:
Odstranjeni: 8, 7, 5
Seznam po `push(4)`: 8, 5, 3, 1, 4

b) (10) Kakšna je v splošnem (najslabša) časovna zahtevnost vsake `push` in `pop` operacije? Koliko dodatnega prostora potrebujemo za izvedbo vsake `push` in `pop` operacije? Konstruiraj zaporedje n `push` operacij, ki trajajo skupaj $\Omega(n \log n)$ časa. Konstruiraj novo zaporedje n `push` operacij, tako da naslednjih n `pop` operacij traja $\Omega(n \log n)$ časa. Odgovore ustrezno utemelji.

Rešitev: Obe operaciji imata časovno zahtevnost $O(\log n)$, kjer je n št. elementov v kopici, in prostorsko zahtevnost $O(1)$.
Če vzamemo zaporedje $1, 2, 3, \ldots, n$ bo vsak novo vstavljeni element moral splacati na vrh kopice, kar bo za i-ti element vzelo $\log i$ korakov. Lahko je videti, da je vsota $\sum_{i=1}^{n} \log i = \Omega(n \log n)$.
Ekvivalenten razmislek lahko naredimo npr. za zaporedje $n, n-1, \ldots, 1$ in za pop operacijo.

2. naloga (20 točk)

Ukvarjamo se z menjavo valut. Znan je uradni referenčni tečaj r, ki za par valut i in j pove, da za a enot valute i dobimo $a \cdot r_{ij}$ enot valute j. Za referenčni tečaj seveda velja $r_{ij}r_{jk}r_{ki} = 1$. Poleg tega vsaka banka kupuje in prodaja valute in po primerjavi različnih tečajev si si ustvaril tabeli b_{ij} in t_{ij}, ki povesta, da valuto i najugodnejše pretvorimo v valuto j v banki b_{ij} po tečaju t_{ij}. Seveda se lahko zgodi, da dveh valut ni možno direktno pretvoriti (kot recimo JPY v CHF spodaj).
Primer:

r	EUR	CHF	JPY
EUR	1	1.14	125.12
CHF	0.88	1	109.66
JPY	0.0080	0.0091	1

t (b)	EUR	CHF	JPY
EUR	/	1.12140 (SKB)	121.12 (Abanka)
CHF	0.8655 (NLB)	/	105.57 (BCGE)
JPY	0.007875 (SKB)	/	/

Od mrzle tete iz daljne dežele si podedoval x enot valute s in bi jo rad najceneje pretvoril v valuto e, pri čemer sploh ni nujno, da obstaja direktne pretvorba, pa tudi najcenejša ni nujno.

a) (5) Prevedi problem v jezik teorije grafov. Natančno definiraj strukturo in problem, ki ga na njej rešuješ.

Rešitev: Vozišča grafa so valute, za izbiro kaj damo na povezave, pa je veliko izbir. Vsekakor je **napačna** izbira, če damo na povezave samo referenčne tečaje, ne da bi uporabili kakršnekoli dejanske tečaje.
Možnost 1: klasična možnost je, da damo na povezave najcenejše dejanske težave in iščemo najdaljšo pot v grafu, kjer uteži množimo. Ker ne maramo produktov, vržemo čez logaritme (t_{ij} so seveda pozitivni), ker ne maramo maksimizacije, damo spredaj minus, in tako rešujemo standarden probem najkrajše poti v grafu s splošnimi utežmi.
Možnost 2: ker imamo na voljo referenčni tečaj, ga lahko uporabimo in povezavo med valutama otežimo z "izgubo" pri menjavi, česar sicer ne bi morali oceniti. Če imamo eno enoto valute v_i pri menjavi v v_j dobimo $t_{i,j}$ valute v_2, kar predstavlja dejansko vrednost $t_{i,j}/r_{i,j}$ v valuti 1. Naš delež izgube je torej $1 - t_{i,j}/r_{i,j}$, s čimer lahko utežimo povezave. Pri tem moramo paziti, da se pri prenosu izgube pravilno seštevajo. Če naredimo menjavo z izgubo p, in nato še eno z izgubo q, je skupna izguba $p + q - pq$, označimo to z $p \oplus q$. Iščemo torej najkrajšo pot v grafu z danimi uteži, pri čemer seštevanje nadomestimo s $\oplus$.

b) (10) Za reševanje tega problema ste na predavanjih spoznali znan algoritem. Izpeljite pogoje, pri katerih ta algoritem smete uporabiti. Ocenite, ali so ti pogoji v realnosti ponavadi izpolnjeni.

Rešitev: Pripada možosti 1: Za to uporabimo Bellman-Fordov algoritem. Pogoj za to je, da nimamo negativnih ciklov, kar se v našem primeru prevede v cikel, kjer bi z menjanjem valut dobili več denarja kot prej. V praksi se to ne dogaja (razen v redkih primerih, ko se nekdo zmoti...) in če bi se zgodilo, bi lahko imeli teoretično neomejen zaslužek.
Pripada možnosti 2: Uporabiti želimo Dijksrov algoritem. Pogoj, da so uteži pozitivne, velja, ko razdalje seštevamo, saj mora za razdaljo d in utež w veljati, da prek povezave ne moremo priti bližje kot smo bili prej, torej $d + w \geq d$ oz. $w \geq 0$. Če bi množili, mora vejati $dw \geq d$ oz. $w \geq 1$, v našem primeru pa velja $d \oplus w \geq d$, kar se prevede v $d + w - dw \geq d$, oz. $w(1 - d) \geq 0$, od koder sledi, da mora biti $w \geq 0$ in $d \leq 1$. da bi bili obe števili negativni, nima smisla. Prvi pogoj pomeni, da mora biti $1 - t_{i,j}/r_{i,j} \geq 0$, oz. $t_{i,j} \leq r_{i,j}$, kar je močnejši pogoj kot pri Bellman-fordu in tudi ta v praksi velja, ker banke pri menjavah ne posujejo z izgubo.

c) (5) Opiši katere spremembe bi bilo treba v algoritmu narediti, da bi kot rezultat vrnil, v katerih bankah je potrebno naredili katero zamenjavo in v kakšnem vrstnem redu ter koliko odstotkov izgube imamo na koncu.
Hranimo podatke o predhodniku. Tako lahko rekonstruiramo pot, nakar imamo vse podatke o tečajih in bankah in lahko z nekaj deljenja in množenja izračunamo našo izgubo.

3. naloga (20 točk)

Dan je seznam n celih števil. Izračunati želimo večino seznama, tj. element, ki se v seznamu pojavi več kot $\frac{n}{2}$-krat, ali pa sporočiti, da tak element ne obstaja.

a) (5) Naj bodo števila v seznamu omejena na interval $[0, M-1]$. Zapiši algoritem in analiziraj časovno in prostorsko zahtevnost.

Rešitev: Ker so elementi števila, si priravimo seznam c velikosti M, ki je na začetku 0. Za vsak element c začetnega seznama nato vrednost $c[x]$ povečamo za 1. Na koncu (ali pa že sproti) gremo še enkrat čez c in če obstaja tak x, da je $c[x] > n/2$, vrnemo x, sicer pa večina ne obstaja.

b) (5) Sprostimo predpostavke glede lastnosti elementov. Elementi niso nujno števila, ampak zgolj objekti, ki pa jih znamo med seboj primerjati glede na neko linearno urejenost $\leq$. Razvij $O(n \log n)$ algoritem za reševanje problema, ki porabi $O(1)$ dodatnega prostora.

Rešitev: Ker imamo na voljo primerjanje elementov, jih lahko uredimo z nekim inplace $O(n \log n)$ sortirnim algoritmom, npr. heapsort. Nato procesiramo elemente od leve proti desni in vzdržujemo števec elementov, ki so enaki trenutnemu. Če ta kdaj preseže $n/2$, vrnemo večino.
Alternativno lahko opazimo, da se večina, če obstaja, gotovo nahaja nasredini seznama. Nato za ta element samo preverimo, ali je res večina.

c) (5) Še bolj sprostimo predpostavke glede lastnosti elementov. Denimo, da znamo primerjati objekte le za enakost. Razvij $O(n^2)$ algoritem za reševanje problema, ki ne porabi nič dodatnega prostora.

Rešitev: Za vsak element seznama preverimo, kolikorat se pojavi noter, tako da gremo še enkrat čez cel seznam.

d) (5) Še vedno imejmo samo elemente, ki jih znamo primerjati za enakost. Razvij $O(n)$ algoritem, ki uporabi le dve dodatni spremenljivki in $O(1)$ dodatnega prostora.
Namig: najprej predpostavi, da večinski element obstaja in ga najdi, nato pa preveri, ali najden element ustreza.

Rešitev: Ta naloga je težja. Algoritem se imenuje *Boyer–Moore majority vote algorithm* in je zelo eleganten in neočiten.
Naj bo x prvi element seznama, ki predstavlja "treutnega kandidata za večino" in $c = 1$, ki predstavlja "trenutno prevlado konadidata" (in ne koliko jih je). Nato se sprehodimo po seznamu naprej: če je trenutni element y enak x, povečamo c za ena, sicer ga zmanjšamo za 1. Če je $c = 0$, nastavimo $x = y$ in nato $c = 1$. Na koncu še preverimo, če je x res večina. Pravilnost ni popolnoma očitna in je zanimiva vaja.

4. naloga (20 točk)

Dane so matrike A_i, velikosti $m_i \times n_i$, za $i = 1, \ldots, N$. Izračunati želimo produkt $P = \prod_{i=1}^{N} A_i$.

a) (5) Kateri pogoji morajo veljati, da je produkt sploh definiran? Koliko je časovna zahtevnost, če izračunamo produkt od leve prosti desni s trivialnim matričnim množenjem?

Rešitev: Matrike morajo biti primernih dimenzij, veljati mora $m_i = n_{i+1}$.
Produkt matrike $m \times p$ z $p \times n$ matriko stane $O(mnp)$ časa. V našem primeru množimo od leve proti desni in imamo $N-1$ produktov. Prvi produkt stane $m_1 n_1 n_2$ in dobimo $m_1 \times n_2$ matriko. Naslednji produkt stane $m_1 n_2 n_3$ in dobimo $m_1 \times n_3$ matriko, itd...
Skupna cena je torej $\sum_{i=1}^{N-1} m_1 n_i n_{i+1}$, kar lahko bolj grobo ocenimo z $O(Nn^3)$, kjer je $n = \max_i\{n_i, m_i\}$.

b) (10) Asociativnost množenja matrik nam dovoljuje, da postavimo oklepaje kakor želimo. Toda ne moremo preveriti vseh možnosti, saj jih je eksponentno mnogo. Razvij algoritem, ki izračuna optimalno postavitev oklepajev, tj. tako postavitev, ki minimizira število potrebnih operacij.
Problem rešujemo z dinamičnim programiranjem. Definiramo si

$$c(i,j) = \text{najmanjša cena za izračun produkta } \prod_{k=i}^{j} A_k.$$

Velja $c(i,i) = 0$ za vsak i in rekurzivna zveza, ki obravnava možne postavitve zadnjega množenja:

$$c(i,j) = \min_{i \le k \le j} \{c(i,k) + c(k+1,j) + m_i n_k n_j\}$$

Izračun optimalenga produkta stane $O(N^3)$.

c) (5) Oglejmo si produkt $x^\mathsf{T} A^k y$ za $x, y \in \mathbb{R}^n$ in $A \in \mathbb{R}^{n \times n}$. Koliko operacij potrebujemo, če produkt izračunamo z algoritmom iz prejšnje točke? Razvij boljši algoritem za velike k z uporabo hitrega potenciranja. Kakšna je njegova časovna zahtevnost? Če predpostavimo, da je konstanta, skrita v O notaciji, enaka 1, za katere n pri danem k uporabimo en in za katere drug algoritem?

Rešitev: Na roke ugotovimo, da je optimalno množiti od leve proti desni (ali pa ravno obratno). Tako vsakič množimo matriko in vektor, kar stane $O(n^2)$ časa. Teh množenj je k, nato pa še en skalarni produkt, ki je zanemarljiv, skupaj $O(kn^2)$ časa.
Pri hitrem potenciranju lahko potenco A^k izračunamo v $\log k$ časa, pri čemer moramo upoštevati, da vsako množenje traja $O(n^3)$. Nato imamo še produkt matrike z vektorjem in skalarni produkt, ki sta zopet zanemarljiva, kar da skupno časovno zahtevnost $O(n^3 \log k)$.
Primerjava je torej sledeča. Enakost

$$kn^2 = n^3 \log k$$

velja za $n = k/\log k$, pri čemer se za n večji od tega k splača uporabiti prvi algoritem. Npr. za $k = 1024$ mora biti $n \le 102$, da se spača uporabiti hitro potenciranje.