

# Algoritmi in podatkovne strukture 2

## Sprotni algoritmi

Luka Fürst

## »Običajni« problemi in algoritmi

- Pri številnih problemih je celoten vhod znan vnaprej
- Optimalno rešitev lahko praviloma poiščemo s preverjanjem vseh možnih rešitev, le da za tak pristop ponavadi potrebujemo preveč časa
- Na primer, najkrajšo pot med dvema vozliščema v grafu bi lahko našli tako, da bi poiskali vse možne poti in izbrali najkrajšo
- Algoritmi nam pomagajo optimalno rešitev najti učinkoviteje

## Sprotni problemi in algoritmi

- Pri nekaterih problemih pa vhod dobivamo **sproti** (»po koščkih«)
- Optimalna rešitev bi zahtevala »**jasnovidnost**« (sposobnost gledanja v prihodnost)
- **Sprotni (online) algoritem**: algoritem, ki se odloča samo na podlagi že prispelih podatkov
- Zanimalo nas bo delovanje sprotnega algoritma v primerjavi s hipotetičnim optimalnim algoritmom, ki ima vedno na voljo celoten tok podatkov

## Vzdrževanje pomnilniških blokov v predpomnilniku

- Predpomnilnik je manjši, a hitrejši od glavnega predpomnilnika, zato bi radi, da se iskani podatek (oz. blok podatkov) čimvečkrat nahaja v predpomnilniku
- Ko dostopamo do bloka, ga prenesemo v predpomnilnik, če ga tam še ni
- Če je ob tej operaciji predpomnilnik poln, moramo enega od blokov izvreči iz predpomnilnika
- Optimalni algoritem: izvrzi blok, ki ga najdlje ne boš znova potreboval (FFU — *farthest-in-future*)
- Ta algoritem je v praksi neizvedljiv, saj ne poznamo zaporedja blokov v prihodnosti
- Zato v praksi uporabljamo (hevrstične) sprotne algoritme (npr. LRU — *least recently used*)

## Stopnice ali dvigalo?

- Iz pritličja bi radi prispeli do  $k$ -tega nadstropja
- Pot po stopnicah zahteva 1 minuto na nadstropje (skupaj torej  $k$  minut)
- Dvigalo potrebuje 1 minuto za celotno pot, a ga bomo (morda) morali čakati
- Kakšen algoritem ubrati, da ne bo bistveno slabši od optimalnega jasnovidnega algoritma, pri katerem točno vemo, koliko časa bomo čakali na dvigalo?

## Optimalen jasnovidni algoritem

- Recimo, da nam »prerok« pove, da bomo na dvigalo čakali natanko  $d$  minut
- Koliko časa bomo porabili za pot do nadstropja  $k$  pri eni ali drugi odločitvi v odvisnosti od  $d$ ?
- Recimo, da je  $k = 5$

| $d$ | Dvigalo | Stopnice | Odločitev |
|-----|---------|----------|-----------|
| 0   | 1       | 5        | dvigalo   |
| 1   | 2       | 5        | dvigalo   |
| 2   | 3       | 5        | dvigalo   |
| 3   | 4       | 5        | dvigalo   |
| 4   | 5       | 5        | vseeno    |
| 5   | 6       | 5        | stopnice  |
| 6   | 7       | 5        | stopnice  |
| ... | ...     | ...      | stopnice  |

## Optimalen jasnoviden algoritem

- Če je  $d \leq k - 2$ , počakaj na dvigalo
- Če je  $d \geq k$ , se povzpni po stopnicah
- Če je  $d = k - 1$ , sta obe odločitvi enakovredni
- Skupna poraba časa za pot do nadstropja  $k$ :

$$J(d) = \begin{cases} d + 1 & \text{pri } d \leq k - 1 \\ k & \text{pri } d \geq k \end{cases} = \min(d + 1, k)$$

# Sprotni algoritmi

- Če ne vidimo v prihodnost, moramo ubrati enega od **sprotnih** **algoritmov**
- **Algoritem 1: »vedno stopnice«**
  - mahnemo jo po stopnicah, tudi če nas dvigalo že čaka
- **Algoritem 2: »vedno dvigalo«**
  - počakamo na dvigalo, ne glede na to, kako dolgo ga bomo čakali

## Konkurenčno razmerje (*competitive ratio*)

- Mera za učinkovitost ali kakovost sprotnega algoritma
- Za kolikšen faktor je **v najslabšem primeru** sprotni algoritem  $S$  slabši od optimalnega jasnovidnega algoritma  $J$ ?

$$c = \max \left\{ \frac{S(I)}{J(I)} \mid I \in \mathcal{I} \right\}$$

- $A(I)$ : vrednost (npr. poraba časa), ki jo proizvede algoritem  $A$  na vходу  $I$
- $\mathcal{I}$ : nabor vseh možnih vhodov
- Če ima sprotni algoritem konkurenčno razmerje  $c$ , je  **$c$ -konkurenčen**

# Algoritem 1

- Kolikšno je konkurenčno razmerje algoritma »vedno stopnice«?
- $\mathcal{I} = \{0, 1, \dots, D - 1\}$  je množica vseh mogočih čakalnih časov na dvigalo
  - recimo, da dvigala nikoli ne bomo čakali več kot  $D$  minut
- $c = \max\{S(d) / J(d) \mid d \in \{0, \dots, D\}\}$

# Algoritem 1

- $c = \max\{S(d) / J(d) \mid d \in \{0, \dots, D\}\}$
- $S(d) = k$  za vsak  $d$
- $J(d) = \min\{d + 1, k\}$
- $c = \max\left\{\frac{k}{1}, \frac{k}{2}, \dots, \frac{k}{k}\right\}$
- $c = k$
- Algoritem »vedno stopnice« je torej  $k$ -konkurenčen

## Algoritem 2

- $c = \max\{S(d) / J(d) \mid d \in \{0, \dots, D\}\}$
- $S(d) = d + 1$
- $J(d) = \min\{d + 1, k\}$
- $c = \max\left\{\frac{1}{1}, \frac{2}{2}, \dots, \frac{k}{k}, \frac{k+1}{k}, \dots, \frac{D+1}{k}\right\}$
- $c = (D + 1) / k$
- Algoritem »vedno dvigalo« je torej  $((D + 1) / k)$ -konkurenčen

## Stopnice ali dvigalo?

- Kateri algoritem je boljši?
- Algoritem »vedno stopnice« je boljši, če je  $D + 1 > k^2$
- Algoritem »vedno dvigalo« je boljši, če je  $D + 1 < k^2$
- Obstaja pa še »špekulativni« algoritem: nekaj časa čakamo na dvigalo, po preteku tega časa pa se povzpne po stopnicah
- Recimo, da čakamo ravno  $k$  minut (toliko, kot traja pot po stopnicah)

# Špekulativni algoritem

- $T(d)$ : čas potovanja s špekulativnim algoritmom, če na dvigalo čakamo  $d \in \{0, \dots, D\}$  minut
- Če je  $d < k$ 
  - $T(d) = d + 1$  ( $d$  minut čakanja, 1 minuta vožnje)
  - $J(d) = d + 1$  (dvigalo)
- Če je  $d = k$ 
  - $T(d) = k + 1$  ( $k$  minut čakanja, 1 minuta vožnje)
  - $J(d) = k$  (stopnice)
- Če je  $d > k$ 
  - $T(d) = 2k$  ( $k$  minut čakanja,  $k$  minut hoje)
  - $J(d) = k$  (stopnice)

# Špekulativni algoritem

$$c = \max \left\{ \frac{1}{1}, \frac{2}{2}, \dots, \frac{k}{k}, \frac{k+1}{k}, \frac{2k}{k} \right\}$$
$$c = 2$$

- Špekulativni algoritem je 2-konkurenčen
- Konkurenčno razmerje je neodvisno od  $k$  ( $c = O(1)$ ), zato je špekulativni algoritem po kriteriju najslabšega scenarija najboljša izbira
- V najslabšem primeru se bo špekulativni algoritem najbolje obnesel, seveda pa to nikakor ne pomeni, da se bo najbolje obnesel tudi v najboljšem ali povprečnem primeru

# Predpomnilnik

- Naj bo  $B = \langle b_1, b_2, \dots, b_n \rangle$  zaporedje zahtev po pomnilniških blokih
  - najprej zahtevamo podatek v bloku  $b_1$ , nato podatek v bloku  $b_2$  itd.
- V predpomnilniku je prostora za  $k < n$  blokov
- Predpomnilnik je na začetku prazen
- Če je blok  $b_i$  že v predpomnilniku, imamo **zadetek** in ne naredimo ničesar
- Če bloka  $b_i$  še ni v predpomnilniku, imamo **zgrešitev**
  - če je predpomnilnik poln, izvržemo enega od blokov v njem
  - blok  $b_i$  prenesemo v predpomnilnik
- Kateri blok izvreči, da bomo imeli karseda malo zgrešitev?

# Optimalen jasnovidni algoritem

- FFU (*farthest-in-future*)
- Izvrzi blok, na katerega se boš naslednjič skliceval najdlje v prihodnosti (oz. sploh nikoli več)
- Takšna strategija je dokazano optimalna (gl. Cormen *et al.*), seveda pa v praksi ne vemo, kakšno zaporedje zahtev nas čaka

## Primer delovanja FFU

- $k = 3$
- $B = \langle 2, 4, 2, 1, 3, 5, 4, 1, 2 \rangle$

| Zahteva | Zgrešitev? | Izvrzi blok ... | Predpomnilnik             |
|---------|------------|-----------------|---------------------------|
| 2       | da         |                 | $\langle 2 \rangle$       |
| 4       | da         |                 | $\langle 2, 4 \rangle$    |
| 2       | ne         |                 | $\langle 2, 4 \rangle$    |
| 1       | da         |                 | $\langle 2, 4, 1 \rangle$ |
| 3       | da         | 2               | $\langle 3, 4, 1 \rangle$ |
| 5       | da         | 3               | $\langle 5, 4, 1 \rangle$ |
| 4       | ne         |                 | $\langle 5, 4, 1 \rangle$ |
| 1       | ne         |                 | $\langle 5, 4, 1 \rangle$ |
| 2       | da         | 1               | $\langle 5, 4, 2 \rangle$ |

## Primeri praktičnih algoritmov

- **FIFO** (*first-in-first-out*): izvrzi blok, ki v predpomnilniku prebiva najdlje
- **LIFO** (*last-in-first-out*): izvrzi blok, ki v predpomnilniku ždi najmanj časa (tj. ki si ga nazadnje dodal)
- **LRU** (*least recently used*): izvrzi blok, na katerega se najdlje časa nisi skliceval
- **LFU** (*least frequently used*): izvrzi blok, na katerega si se doslej najmanjkrat skliceval; če je takih blokov več, izvrzi tistega, ki je najdlje prisoten v predpomnilniku

# LIFO

## Trditev

Algoritem LIFO (izvrzi blok, ki si ga v predpomnilnik nazadnje dodal) ima konkurenčno razmerje  $c = \Theta(n/k)$ .

$$c = \max \left\{ \frac{z_{\text{LIFO}}(B)}{z_{\text{FFU}}(B)} \mid B \in \mathcal{B} \right\}$$

- $z_A(B)$ : število zgrešitev, ki jih pri obravnavi zaporedja  $B = \{b_1, \dots, b_n\}$  sproži algoritem  $A$
- $\mathcal{B}$ : celoten nabor možnih zaporedij  $n$  zahtev

# LIFO

- Najprej pokažimo, da je  $c = \Omega(n/k)$
- Oglejmo si sledeče zaporedje dolžine  $n$ :

$$B = \langle 1, 2, \dots, k, k+1, k, k+1, k, k+1, \dots \rangle$$

- Vsaka od prvih  $k$  zahtev v vsakem primeru povzroči zgrešitev, saj je predpomnilnik na začetku prazen
- Tudi vsaka nadaljnja zahteva povzroči zgrešitev
  - ko uvozimo blok  $k+1$ , izvržemo blok  $k$ , saj smo ga nazadnje dodali
  - ko uvozimo blok  $k$ , izvržemo blok  $k+1$ , saj smo ga nazadnje dodali
- LIFO na zaporedju  $B$  torej sproži  $n$  zgrešitev
- Optimalni jasnovidec FFU sproži zgolj  $k+1$  zgrešitev
- $c \geq n/(k+1) = \Omega(n/k)$

# LIFO

- Velja tudi  $c = O(n/k)$
- Noben algoritem, naj bo še tako nespameten, ne more na zaporedju dolžine  $n$  sprožiti več kot  $n$  zgrešitev
- Noben algoritem, naj bo še tako moder, ne more na zaporedju dolžine  $n$  sprožiti manj kot  $k$  zgrešitev, saj je predpomnilnik na začetku prazen
  - seveda to velja tudi za najmodrejšega med modrimi, torej FFU
- $c(\text{LIFO}) = \Theta(n/k)$

# LRU

- Izvrzi blok, na katerega se najdlje nisi skliceval

| Zahteva | Zgrešitev? | Izvrzi blok ... | Predpomnilnik             |
|---------|------------|-----------------|---------------------------|
| 2       | da         |                 | $\langle 2 \rangle$       |
| 4       | da         |                 | $\langle 2, 4 \rangle$    |
| 2       | ne         |                 | $\langle 2, 4 \rangle$    |
| 1       | da         |                 | $\langle 2, 4, 1 \rangle$ |
| 3       | da         | 4               | $\langle 2, 3, 1 \rangle$ |
| 5       | da         | 2               | $\langle 5, 3, 1 \rangle$ |
| 4       | da         | 1               | $\langle 5, 3, 4 \rangle$ |
| 1       | da         | 3               | $\langle 5, 1, 4 \rangle$ |
| 2       | da         | 5               | $\langle 2, 1, 4 \rangle$ |

# LRU

## Trditev

LRU ima konkurenčno razmerje  $O(k)$ .

- Za potrebe analize razdelimo zaporedje  $B$  na epohe 1, 2, 3 itd.
- Epoha  $i$  traja od konca epohe  $i - 1$  do trenutka, ko števec medsebojno različnih zahtev preseže  $k$
- Na začetku vsake epohe nastavimo števec na 0
- Epoke pri  $k = 3$  in  $B = \langle 5, 1, 4, 1, 2, 2, 5, 2, 6, 2, 1, 7, 3, 6, 2, 6 \rangle$ :  
5, 1, 4, 1, 2, 2, 5, 2, 6, 2, 1, 7, 3, 6, 2, 6

# LRU

- Trdimo, da LRU v vsaki epohi sproži kvečjemu  $k$  zgrešitev
- Prvi sklic na blok  $b$  v epohi lahko sproži zgrešitev, naslednji sklici na blok  $b$  v isti epohi pa zagotovo ne bodo sprožili zgrešitve
  - LRU bo v vmesnem obdobju namesto bloka  $b$  izvrgel katerega od blokov iz prejšnje epohe, saj je blok  $b$  med  $k$  bloki z najpoznejšimi časi zadnjega sklica
- Na primer, po epohi 5, 1, 4, 1 je vsebina predpomnilnika 1, 4, 5
- V okviru epohe 2, 2, 5, 2, 6, 2 prvi sklic na blok 2 sproži zgrešitev (izvržemo blok 5), nato pa bo blok 2 ostal v predpomnilniku vsaj do konca epohe
  - ob sklicu na blok 5 bomo izvrgli blok 4
  - ob sklicu na blok 6 bomo izvrgli blok 1

## LRU vs. FFU

- Optimalen jasnovidni algoritem (FFU) v prvi epohi sproži  $k$  zgrešitev
- Cormen *et al.* **napačno** trdijo, da FFU v vsaki nadaljnji epohi sproži najmanj eno zgrešitev (vsaj ob prvi zahtevi v epohi)
- [https://mitp-content-server.mit.edu/books/content/sectbyfn/books\\_pres\\_0/11599/e4-bugs.html](https://mitp-content-server.mit.edu/books/content/sectbyfn/books_pres_0/11599/e4-bugs.html)
- Dejansko imamo v **dveh zaporednih** epohah najmanj eno zgrešitev, saj imamo vsaj  $k + 1$  medsebojno različnih zahtev
- FFU torej sproži v povprečju najmanj  $1/2$  zgrešitve na epoho
- LRU sproži kvečjemu  $k$  zgrešitev na epoho
- Konkurenčno razmerje je torej kvečjemu  $k/(1/2) = O(k)$

## LRU vs. LIFO

- $c(\text{LIFO}) = O(n/k)$
- $c(\text{LRU}) = O(k)$
- Algoritem LRU je bistveno boljši, saj je njegovo konkurenčno razmerje neodvisno od dolžine zaporedja zahtev ( $n$ )
- Ponovno velja opozoriti, da delamo analizo za **najslabši** primer
- V najboljšem ali morda celo povprečnem primeru bo lahko LIFO čisto spodobno deloval, v najslabšem primeru pa je LRU nesporno boljši, saj nikoli ne sproži več kot  $\Theta(k)$ -krat toliko zgrešitev kot optimalni jasnovidec

# Splošna spodnja meja konkurenčnega razmerja

## Trditev

Vsak deterministični sproti algoritem za rokovanje s predpomnilnikom ima konkurenčno razmerje  $\Omega(k)$ .

- Kolikšno je razmerje med številom zgrešitev, ki jih sproži poljuben sproti algoritem, in številom zgrešitev, ki jih sproži algoritem FFU, v scenariju, ki je za sproti algoritem najmanj ugoden?
- Naj bo velikost predpomnilnika enaka  $k$
- Recimo, da imamo  $n$  zahtev in  $k + 1$  medsebojno različnih blokov
- Vsakemu sprotnemu algoritmu lahko »nagajamo« tako, da mu vsakokrat serviramo zahtevo po bloku, ki ga je ravnokar izvrigel
- Na ta način bomo sprožili  $n$  zgrešitev v  $n$  zahtevah

## Splošna spodnja meja konkurenčnega razmerja

- Recimo, da je predpomnilnik v nekem trenutku poln
- To pomeni, da vsebuje  $k$  medsebojno različnih blokov; ostane torej samo še en blok (npr.  $b \in \{1, \dots, k + 1\}$ )
- Ob zahtevi po bloku  $b$  se bo torej sprožila zgrešitev
- Algoritem FFU vidi v prihodnost in bo izvrigel blok, na katerega vsaj v naslednjih  $k$  zahtevah ne bo nobenega sklica
- To pomeni, da se bo naslednja zgrešitev zgodila najmanj  $k + 1$  iteracij kasneje
- Ob naslednji zgrešitvi se logika ponovi
- V  $n$  iteracijah bo algoritem FFU torej sprožil kvečjemu  $k + (n - k) / k$  zgrešitev (največ  $k$  se jih sproži takoj na začetku)

# Splošna spodnja meja konkurenčnega razmerja

- Poljuben sprotni algoritem:  $n$  zgrešitev za zaporedje dolžine  $n$
- Optimalen jasnovidni algoritem: kvečjemu  $k + (n - k) / k$  zgrešitev pri istem zaporedju

$$c = \frac{n}{k + (n - k) / k} = \frac{nk}{n + k^2 - k}$$

- Pri dovolj velikih  $n$  je  $c \approx nk/n = k$
- Ker je v imenovalcu »kvečjemu«, je  $c = \Omega(k)$

## Preurejanje povezanega seznama

- Podan je dvosmerno povezan seznam  $L$  z medsebojno različnimi elementi in zaporedje zahtev po elementih ( $Z$ )
- Naj bo  $r_L(x)$  položaj (indeks) elementa  $x$  v seznamu  $L$ 
  - indeksi se pričnejo z 1
- Za dostop do elementa  $x$  v seznamu  $L$  potrebujemo  $r_L(x)$  časovnih enot ( $\text{cena} = r_L(x)$ )
- Če pričakujemo, da bomo pogosto dostopali do določenega elementa, se nam ga splača prestaviti na začetek seznama
- Za prestavitev elementa  $x$  na začetek potrebujemo  $r_L(x) - 1$  časovnih enot (toliko zaporednih zamenjav opravimo)

# Primer

- Naj bo  $L = \langle 10, 20, 30, 40, 50 \rangle$
- Zaporedje zahtev naj bo  $Z = \langle 50, 30, 40, 40 \rangle$
- Brez preurejanja je skupna cena iskanja enaka  $5 + 3 + 4 + 4 = 16$
- Če po odkritju elementa 30 prestavimo element 40 na začetek:
  - cena 5 za iskanje elementa 50
  - cena 3 za iskanje elementa 30
  - cena 1 za pot do elementa 40
  - cena 3 za prestavitev elementa 40 na začetek (element 40 po vrsti zamenjamo z elementi 30, 20 in 10)
  - cena 1 za iskanje elementa 40
  - cena 1 za iskanje elementa 40
  - skupna cena = 14

## Jasnovidni in sprotni algoritem

- Jasnovidni algoritem  $J$  (pred)vidi celotno zaporedje zahtev in z vnaprejšnjim prestavljanjem elementov na začetek minimizira skupno ceno iskanj in prestavljanj
- Sprotni algoritem  $S$  pa bomo definirali tako, da bo vedno premaknil pravkar iskani element na začetek
  - element, ki smo ga pravkar iskali, bomo morda vnovič iskali v bližnji prihodnosti
  - če ga bomo vnovič iskali po  $k \ll n$  iteracijah, bomo na ta način lahko prihranili precej časa
- Kolikšno je v najslabšem primeru na nekem zaporedju zahtev razmerje med porabo časa algoritma  $S$  in porabo časa algoritma  $J$ ?

## Cena iskanja s sprotnim algoritmom

- Naj bo  $L_0^S = L$ ,  $L_1^S$ ,  $L_2^S$  itd. zaporedje seznamov, ki jih tvorijo posamezne iteracije sprotnega algoritma
  - v  $i$ -ti iteraciji obdelamo  $i$ -to zahtevo (poiščemo pripadajoči element) in iskani element premaknemo na začetek
- Na primer, če je  $L = \langle 10, 20, 30, 40, 50 \rangle$  in  $Z = \langle 50, 30, 40, 40 \rangle$ , dobimo

$$L_0^S = \langle 10, 20, 30, 40, 50 \rangle$$

$$L_1^S = \langle 50, 10, 20, 30, 40 \rangle$$

$$L_2^S = \langle 30, 50, 10, 20, 40 \rangle$$

$$L_3^S = \langle 40, 30, 50, 10, 20 \rangle$$

$$L_4^S = \langle 40, 30, 50, 10, 20 \rangle$$

## Cena iskanja s sprotnim algoritmom

- Sprotni algoritem v  $i$ -ti iteraciji v seznamu  $L_{i-1}$  poišče element  $x$  in za to porabi  $r_{L_{i-1}^S}(x)$  enot časa
- Po odkritju ga premakne na začetek seznama in za to porabi  $r_{L_{i-1}^S}(x) - 1$  enot časa (toliko je zamenjav s sosednjimi elementi)
- Skupna poraba časa (cena) sprotnega algoritma v  $i$ -ti iteraciji je potemtakem  $2r_{L_{i-1}^S}(x) - 1$

## Cena iskanja z jasnovidnim algoritmom

- Naj bo  $L_0^J = L, L_1^J, L_2^J$  itd. zaporedje seznamov, ki jih tvorijo posamezne iteracije optimalnega jasnovidnega algoritma
- Jasnovidni algoritem v  $i$ -ti iteraciji v seznamu  $L_{i-1}$  poišče element  $x$  in za to porabi  $r_{L_{i-1}^J}(x)$  enot časa
- Po odkritju elementa  $x$  izvede neznano število optimizacijskih korakov (pomikov po seznamu in medsebojnih zamenjav elementov)
- Označimo to število s  $t_i$
- Skupna poraba časa (cena) jasnovidnega algoritma v  $i$  iteraciji je potemtakaem  $r_{L_i^J}(x) + t_i$

## Primerjava seznamov

- Jasnovidni in sprotni algoritem pričneta z istim seznamom, vendar pa ga potem preurejata vsak na svoj način
- Ne vemo, kako jasnovidni algoritem preureja seznam, kljub temu pa lahko določimo podobnost obeh seznamov
- **Inverzija** med seznamoma  $L$  in  $L'$ , ki vsebujeta istih  $n$  elementov, a v različnem vrstnem redu, je par  $(x, y)$  z lastnostjo  $x < y$ , tako da je  $r_L(x) < r_L(y)$  in  $r_{L'}(x) > r_{L'}(y)$  ali  $r_L(x) > r_L(y)$  in  $r_{L'}(x) < r_{L'}(y)$

## Primerjava seznamov

- $I(L, L')$  je število inverzij med seznamoma  $L$  in  $L'$ 
  - to bo mera podobnosti seznamov
- $I(\langle 20, 50, 30, 10, 40 \rangle, \langle 40, 20, 10, 30, 50 \rangle) = 7$ 
  - inverzije so  $(10, 30)$ ,  $(10, 40)$ ,  $(10, 50)$ ,  $(20, 40)$ ,  $(30, 40)$ ,  $(30, 50)$  in  $(40, 50)$

## Primerjava seznamov

- $L_{i-1}^S$  in  $L_{i-1}^J$  sta seznama, ki ju pridobimo iz seznama  $L$  po  $i - 1$  iteracijah izvajanja sprotnega oz. jasnovidnega algoritma
- Recimo, da v  $i$ -ti iteraciji iščemo element  $x$
- $BB$  naj bo množica elementov, ki se v obeh seznamih nahajajo levo od  $x$
- $BA$  naj bo množica elementov, ki se v seznamu  $L_{i-1}^S$  nahajajo levo od  $x$ , v seznamu  $L_{i-1}^J$  pa desno od  $x$
- $AB$  naj bo množica elementov, ki se v seznamu  $L_{i-1}^S$  nahajajo desno od  $x$ , v seznamu  $L_{i-1}^J$  pa levo od  $x$

## Primerjava seznamov

- $r_{L_{i-1}^S}(x) = |BB| + |BA| + 1$
- $r_{L_{i-1}^J}(x) = |BB| + |AB| + 1$
- Primer
  - $L_{i-1}^S = \langle 20, 50, 30, 10, 40 \rangle$
  - $L_{i-1}^J = \langle 40, 20, 10, 30, 50 \rangle$
  - $x = 30$
  - $BB = \{20\}$
  - $BA = \{50\}$
  - $AB = \{10, 40\}$
  - $r_{L_{i-1}^S}(x) = |BB| + |BA| + 1 = 3$
  - $r_{L_{i-1}^J}(x) = |BB| + |AB| + 1 = 4$

## Sprememba števila inverzij

- Pričnemo s seznamoma  $L_{i-1}^S$  in  $L_{i-1}^J$
- V seznamu  $L_{i-1}^S$  poiščemo element  $x$  na položaju  $r_{L_{i-1}^S}(x)$ , ga prestavimo na začetek in dobimo seznam  $L_i^S$
- Seznam  $L_{i-1}^J$  pustimo zaenkrat pri miru
- Kako se spremeni število medsebojnih inverzij?
- Koliko je  $I(L_i^S, L_{i-1}^J) - I(L_{i-1}^S, L_{i-1}^J)$ ?

## Sprememba števila inverzij

- Element  $x$  po odkritju zaporedoma zamenjamo z vsemi elementi na njegovi levi (torej z vsemi elementi v množici  $BB \cup BA$ )
- Vsaka zamenjava dveh sosednjih elementov poveča ali zmanjša število inverzij za 1

## Sprememba števila inverzij

- Če  $x$  zamenjamo z elementom  $y \in BB$ , se število inverzij poveča za 1
  - $y$  je bil pred zamenjavo v obeh seznamih levo od  $x$ , sedaj pa to je v seznamu  $L_i^S$  desno od  $x$
- Če  $x$  zamenjamo z elementom  $y \in BA$ , se število inverzij zmanjša za 1
  - $y$  je bil pred zamenjavo v seznamu  $L_{i-1}^S$  levo od  $x$ , v seznamu  $L_{i-1}^J$  desno od  $x$ , sedaj pa je v obeh seznamih desno od  $x$
- $I(L_i^S, L_{i-1}^J) - I(L_{i-1}^S, L_{i-1}^J) = |BB| - |BA|$

# Konkurenčno razmerje sprotnega algoritma

Trditev

Sprotni algoritem je 4-konkurenčen.

# Amortizacijska analiza (kratka ponovitev)

- Spomnimo se na **amortizacijsko analizo**
- Analiziramo zaporedje operacij, ki jih izvajamo na neki podatkovni strukturi
- **Računovodska cena**  $i$ -te operacije je vrednost  $\hat{c}_i$ , ki jo uporabljamo namesto dejanske cene  $c_i$  za lažjo analizo
- No, računovodsko ceno lahko namesto dejanske uporabljamo le v primeru, če za vsako zaporedje operacij velja  $\sum_i \hat{c}_i \geq \sum_i c_i$

## Amortizacijska analiza (kratka ponovitev)

- **Potencial**  $\Phi_i$  predstavlja sposobnost plačevanja za prihodnje operacije po  $i$  izvedenih operacijah na podatkovni strukturi
- Če začetni potencial nastavimo na 0 in če zagotovimo, da potencial nikoli ne pade pod 0, potem nam potencial določa računovodsko ceno

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}$$

- Vrnimo se na analizo našega sprotnega algoritma ...

## Definicija potenciala

- Naša podatkovna struktura je po  $i$ -ti iteraciji sestavljena iz seznamov  $L_i^S$  in  $L_i^J$
- Potencial definirajmo kot

$$\Phi_i = 2I(L_i^S, L_i^J)$$

- $\Phi_0 = 0$ ,  $\Phi_i \geq 0$  za vsak  $i$
- Potencial torej ustreza zahtevam in smiselno zajema podobnost med seznamoma, toda ... zakaj  $2I(L_i^S, L_i^J)$ , ne pa preprosto  $I(L_i^S, L_i^J)$ ?
- Odgovor: »magija«, brez katere se dokaz ne izide

## Potencial in računovodska cena

- Računovodska cena  $i$ -te iteracije sprotnega algoritma:

$$\hat{c}_i^S = c_i^S + \Phi_i - \Phi_{i-1}$$

- Dejanska cena je (kot že vemo)

$$c_i^S = 2r_{L_{i-1}^S}(x) - 1$$

- Koliko pa je  $\Phi_i - \Phi_{i-1} = 2(I(L_i^S, L_i^J), I(L_{i-1}^S, L_{i-1}^J))$ ?

$$\Phi_i - \Phi_{i-1}$$

- Sprotni algoritem poveča potencial za natanko  $2(|BB| - |BA|)$
- Jasnovidni algoritem opravi največ  $t_i$  zamenjav
- Vsaka zamenjava poveča ali zmanjša potencial za 2 (uvede novo ali odstrani obstoječo inverzijo)
- Jasnovidni algoritem torej poveča potencial za kvečjemu  $2t_i$
- Torej je

$$\Phi_i - \Phi_{i-1} \leq 2(|BB| - |BA|) + 2t_i$$

## Še malo algebraične telovadbe ...

$$\begin{aligned}\hat{c}_i^S &= c_i^S + \Phi_i - \Phi_{i-1} \\ &\leq 2r_{L_{i-1}^S}(x) - 1 + 2(|BB| - |BA| + t_i) \\ &= 2r_{L_{i-1}^S}(x) - 1 + 2(|BB| - (r_{L_{i-1}^S}(x) - 1 - |BB|) + t_i)\end{aligned}$$

- Juhuhu! Člena  $2r_{L_{i-1}^S}(x)$  odpadeta!
- Zdaj je jasno, čemú faktor 2 v potencialu ...

$$\begin{aligned}\hat{c}_i^S &\leq 4|BB| + 1 + 2t_i \\ &\leq 4|BB| + 4|AB| + 4 + 4t_i \text{ (magije ni nikoli preveč ...)} \\ &= 4(|BB| + |AB| + 1 + t_i) \\ &= 4(r_{L_{i-1}^J}(x) + t_i) \\ &= 4c_i^J\end{aligned}$$

# Veliki finale

- Ker smo potencial ustrezno definirali, za vsak  $i$  velja

$$\sum_i \hat{c}_i^S \geq \sum_i c_i^S$$

- Torej je

$$\sum_i c_i^S \leq \sum_i \hat{c}_i^S \leq \sum_i 4c_i^J = 4 \sum_i c_i^J$$

- Sprotni algoritem je torej **4-konkurenčen**, in to smo ugotovili, ne da bi sploh vedeli, kako deluje jasnovidni algoritem!
- Na poljubnem zaporedju bo torej sprotni algoritem porabil kvečjemu 4-krat toliko časa kot optimalni jasnovidni algoritem