Nekaj osnovnih algoritmov na seznamih in nizih

Janez Brank

Urejanje

Urejanje

- Imamo seznam/tabelo $a[0..n-1]$, radi bi jo uredili
  - Elemente seznama hočemo prerazporediti tako, da bo za vsak i veljalo $a[i-1] < a[i]$
  - V splošnem si znak „<“ tukaj razlagajmo takole:
    „$x < y$“ = „v vrstnem redu, po katerem želimo urediti naš seznam, mora priti x prej kot y“
  - Mnogi algoritmi za urejanje ne predpostavijo o podatkih nič drugega kot to, da se jih dá primerjati glede na relacijo <
- Notranje vs. zunanje urejanje
  - Ali imamo naključen dostop do podatkov ali le zaporednega?
- Stabilno urejanje
  - Če je več elementov enakih (glede na <), ali ostane njihov relativni medsebojni vrstni red nespremenjen?

Bubble sort

- Primerjamo po dva zaporedna elementa, po potrebi ju zamenjamo (*swap*):

  for ($i = 1$; $i < n$; i++)

  if ($a[i] < a[i-1]$) *swap*($a[i-1]$, $a[i]$);

- Gornji postopek ponavljamo, dokler je še kaj zamenjav
  - Po k izvedbah tega postopka je največjih k elementov že na pravih mestih na koncu tabele

```
for (neurejeni = n ; neurejeni > 1; neurejeni--)
    for (i = 1; i < neurejeni ; i++)
        if (a[i] < a[i – 1]) swap(a[i – 1], a[i]);
```

Urejanje z vstavljanjem (insertion sort)

- Recimo, da imamo v $a[0..u-1]$ že urejen del seznama, zdaj bi radi vanj vrinili element $a[u]$

```
novi = a[u]; i = u;
while (i > 0 && novi < a[i – 1]) {
    a[i] = a[i – 1]; i = i – 1; }
a[i] = novi ;
```

- Da uredimo cel seznam, moramo le počasi povečevati u za 1:

```
for (u = 2; u < n; u++)
    Vrini(u) // postopek od zgoraj
```

Urejanje z izbiranjem (selection sort)

- Poiščimo največji element in ga premaknimo na konec tabele (na indeks $n - 1$)
  - Neurejeni del tabele je zdaj za en element krajši, gornji postopek ponovimo na njem

```
for (neurejeni = n ; neurejeni > 1; neurejeni--) {
    for (kjeMax = 0, i = 1; i < neurejeni ; i++)
        if (a[kjeMax] < a[i]) kjeMax = i ;
    swap(a[kjeMax], a[neurejeni – 1]); }
```

Zlivanje (merging)

- Dve urejeni zaporedji bi radi „zlili“ v eno
  Vhod: $a[0..n_A - 1]$ in $b[0..n_B - 1]$
  Izhod: $c[0..n_A + n_B - 1]$
  $i_A = 0$; $i_B = 0$; $i_C = 0$;
  while ($i_A < n_A$ && $i_B < n_B$)
  if ($a[i_A] < b[i_B]$) $c[i_C{+}{+}] = a[i_A{+}{+}]$;
  else $c[i_C{+}{+}] = b[i_B{+}{+}]$;
  while ($i_A < n_A$) $c[i_C{+}{+}] = a[i_A{+}{+}]$;
  while ($i_B < n_B$) $c[i_C{+}{+}] = b[i_B{+}{+}]$;
- Lahko bi šli tudi od konca proti začetku
  - Tedaj lahko za *c* uporabimo isto tabelo kot za *a*
- Do podatkov potrebujemo le zaporedni dostop
  - Koristno, če so v linked listah ali pa v datotekah na disku/traku
- C++: *merge* v <*algorithm*>

Urejanje z zlivanjem (merge sort)

- Rekurzivna ideja:
  - Razbijmo zaporedje na dva kosa
  - Uredimo vsakega posebej (rekurzivni klic)
  - Na koncu ju zlijmo v eno urejeno zaporedje

```
funkcija MergeSort (a[0..n – 1]):
    if (n <= 1) return;
    L = a[0..n/2 – 1]; D = a[n/2..n – 1];
    MergeSort (L); MergeSort (D);
    zlij L in D v a (s postopkom s prejšnje strani);
```

- Časovna zahtevnost: $T(n) = O(n) + 2\,T(n/2)$
  - Iz tega dobimo $T(n) = O(n \log n)$
- Prostorska zahtevnost: $O(n)$ pomožnega prostora za L in D
  - Če imamo podatke v linked listah, pa jih moramo le premikati med njimi – samo $O(\log n)$ pomnilnika (globina rekurzije)

Hitro urejanje (quicksort)

- Spet rekurzivna ideja:
  - Izberimo si poljuben element tabele *a*, recimo mu *m*
  - Razdelimo *a* na levi del (elementi, manjši od *m*) in desni del (elementi, ≥ *m*)
  - Uredimo vsak del posebej z rekurzivnim klicem

```
funkcija Quicksort (l, d):  // mora urediti a[l..d – 1]
    if (d – l < 2) return; // trivialno
    m = eden od elementov a[l ], …, a[d – 1];
    i = l ; j = d;
    while (i < j) {          // "partition"
        // Elementi a[l..i – 1] so < m; elementi a[ j..d – 1] so ≥ m.
        if (a[i] < m) { i++; continue; }
        if (a[j – 1] ≥ m) { j--; continue; }
        swap(a[i], a[ j – 1]); i++; j--; }
    QuickSort (l, i ); QuickSort (j, d);
```

Hitro urejanje (quicksort)

- Časovna zahtevnost: $T(n) = O(n) + T(n') + T(n - n')$, če smo razdelili tabelo dolžine n na levi del dolžine n' in desni del dolžine $n - n'$
  - Najslabši primer: če vedno delimo na 1 : $(n - 1)$, gre rekurzija n nivojev globoko, $T(n) = O(n^2)$
  - Najboljši primer: če delimo na $n/2 : n/2$, gre rekurzija $\log_2 n$ nivojev globoko, $T(n) = O(n \log n)$
  - Povprečje je, kot se izkaže, tudi $O(n \log n)$
- Velikost levega in desnega dela pa je odvisna od tega, kaj smo izbrali za m
  - Smola: če je m največji/najmanjši element
  - Ugodno: če je m nekje bolj na sredi (npr. mediana)
  - Lahko m izberemo naključno
    - Potem je zelo malo verjetno, da bi imeli ves čas smolo
    - Lahko izberemo tri naključne m, uporabimo srednjega od njih
  - Ko je n majhen, preklopimo na insertion sort ipd.

Izbor k-tega najmanjšega elementa

- Imamo neurejeno tabelo $a[0..n-1]$,
  radi bi poiskali k-ti najmanjši element
  - Seveda jo lahko najprej uredimo – toda ali gre brez tega?
  - V enem prehodu čez tabelo lahko poiščemo najmanjši element
    - V naslednjem prehodu drugega najmanjšega itd.
    - Pazimo na primere, če obstaja več enakih elementov
    - $O(k\,n)$ časa za k-ti najmanjši/največji element
  - Boljša ideja: zgledujmo se po quicksortu ("**quickselect**")
    - Razdelimo $a[0..n-1]$ na levi del $a[0..n'-1]$, kjer so vsi elementi $< m$, in desni del $a[n'..n-1]$, kjer so vsi elementi $\geq m$
    - Če je $k < n'$, moramo v nadaljevanju iskati k-ti najmanjši elt. v levem delu
    - Sicer moramo iskati $(k-n')$-ti najmanjši element v desnem delu
  - To je kot quicksort, le da namesto dveh rekurzivnih klicev izvedemo le enega od njiju: $TS(n) = O(n) + \{TS(n') \text{ ali } TS(n-n')\}$
    - Spet, če imamo smolo ($n' = 1$ ali $n' = n-1$), nastane $TS(n) = O(n^2)$
    - V najboljšem primeru (in tudi v povprečju) pa je $TS(n) = O(n)$
  - C++: *nth_element* v <*algorithm*>

Mediana median

- Zanimiv prijem za izbor m-ja pri quicksortu/quickselectu
  - Razdelimo elemente a-ja na skupine po 5
  - V vsaki skupini poiščimo mediano
  - Iz teh median sestavimo nov seznam dolžine $n/5$
  - S quickselectom poiščimo njegovo mediano – to bo naš m
  - Zagotovo je vsaj 30% elementov prvotnega seznama manjših od m in vsaj 30% večjih od m
  - Cena za quickselect je zdaj v najslabšem primeru $TS(n) = O(n) + TS(n/5) + TS(7n/10)$
  - Za quicksort pa $T(n) = O(n) + TS(n/5) + T(3n/10) + T(7n/10)$
  - Izkaže se $TS(n) = O(n)$ in $T(n) = O(n \log n)$, zdaj torej tudi v najslabšem primeru

Urejanje s štetjem (counting sort)

- Recimo, da so elementi *a*-ja majhna naravna št. od 0 do $M - 1$
  - Za vsako možno vrednost preštejmo, kolikokrat se pojavlja (naredimo „histogram“ tabele)
  - Nato lahko urejeno tabelo kar zgeneriramo

```
for (x = 0; x < M; x++) hist[x] = 0;
for (i = 0; i < n; i++) hist[a[i]] += 1;
for (x = 0, i = 0; x < M; x++)
    for (j = 0; j < hist[x]; j++) a[i++] = x;
```

- Tu smo torej predpostavili, da lahko elemente *a*-ja uporabimo kot indekse v tabelo *hist*
  - Če to ne gre, lahko za *hist* uporabimo slovar
  - Ključe tako dobljenega slovarja bomo morali urediti po kakšnem drugem postopku, vendar imamo zdaj vsak ključ samo enkrat, v *a* pa je bil mogoče večkrat

Urejanje s štetjem (counting sort)

- Kaj če so v tabeli pari (ključ, spremljevalni podatki) in urejene tabele ne moremo kar zgenerirati samo iz ključev?

```
for (x = 0; x < M; x++) hist[x] = 0;
for (i = 0; i < n; i++) hist[a[i].ključ] += 1;
for (x = 0, i = 0; x < M; x++) {
    kam[x] = i; i += hist[x]; }
for (i = 0; i < n; i++) {
    x = a[i].ključ; b[kam[x]++] = a[i]; }
```

- Dobili smo primerno urejeno izhodno tabelo *b*
- Takšno urejanje je tudi stabilno
- Z malo pazljivosti lahko *kam* hranimo v isti tabeli kot *hist*

Urejanje z vedri (bucket sort)

- Razdelimo elemente tabele *a* na *B* „veder“ (buckets) tako, da so vsi elementi v prvem vedru manjši od vseh v drugem vedru in tako naprej
  - Uredimo vsako vedro posebej
  - Staknimo rezultate skupaj

```
for (b = 0; b < B; b++) vedro[b] = prazen seznam;
for (i = 0; i < n; i++) {
    x = a[i].ključ; dodaj a[i] v vedro[ f(x)]; }
for (b = 0; b < B; b++) uredi vedro[b];
for (i = 0, b = 0; b < B; b++)
    for (elt : vedro[b]) a[i++] = elt;
```

  - Vprašanje je, ali znamo poceni računati neko primerno funkcijo $f(x)$, ki ključe čim bolj enakomerno razprši med vedra
  - Npr. če so ključi realna števila z območja $[min, max)$, lahko vzamemo $f(x) = floor((x - min) / (max - min) * B)$
  - Idealni scenarij: $B = n$, vsako vedro ima v povprečju en element, zato urejanje posameznega vedra traja $O(1)$, celoten postopek pa $O(n)$

Urejanje po števkah (radix sort)

- Recimo, da so naši ključi k-mestna naravna števila
  - Razdelimo jih na 10 veder glede na najbolj levo števko (*MSD*)
  - Uredimo vsako vedro z enakim postopkom (rekurzivni klic), le da najbolj levo števko takrat odmislimo
  - Staknimo rezultate skupaj
- Ali pa:
  - Uredimo jih s štetjem glede na najbolj desno števko (*LSD*)
  - Nato ta seznam spet uredimo s štetjem glede na predzadnjo števko
    - Urejanje s štetjem je stabilno, zato so zdaj urejeni po zadnjih dveh števkah skupaj
  - Nato ga uredimo s štetjem glede na predpredzadnjo števko itd.
- Porabili smo $O(k\ n)$ časa za urejanje n k-mestnih števil
- To bi znalo priti prav tudi za urejanje nizov po abecedi ipd.

Kopica

Prioritetna vrsta

- Spomnimo se: vrsta (queue) = podatkovna struktura, v katero dodajamo na koncu in brišemo na začetku
  - Brišemo vedno element, ki je najdlje v vrsti
  - FIFO = first in, first out = kdor prej pride, prej melje
- Prioritetna vrsta (priority queue):
  - Vsak element ima „prioriteto“
  - Brišemo vedno element z najvišjo prioriteto
- Najpogostejša implementacija: (binarna) kopica
  - So še druge kopice: b-iške za $b > 2$, binomske kopice, Fibonaccijeve kopice
- Znan primer uporabe: Dijkstrov algoritem za najkrajše poti v grafu

Kopica

- Kopica je binarno drevo, v katerem:
  - Vsi nivoji so polni, razen mogoče zadnjega
  - Na zadnjem nivoju so vsa vozlišča na levi strani
  - Vsak element ima vsaj tolikšno prioriteto kot njegova otroka
    - Posledica: koren ima najvišjo prioriteto med vsemi
    - Koristno za brisanje elementa z najvišjo prioriteto – vemo, kje ga dobiti
- Kopica z n elementi ima torej povsem predvidljivo obliko
  - Primerna za implementacijo s tabelo $a[0..n-1]$
  - Vozlišče i ima otroka $2i + 1$ in $2i + 2$ ter starša $\lfloor (i-1) / 2 \rfloor$
  - Vozlišče i je notranje za $i < \lfloor n/2 \rfloor$, sicer je list
  - Vozlišče 0 je koren
  - Kopica ima $1 + \lfloor \log_2 n \rfloor$ nivojev
- C++: *make_heap*, *push_heap*, *pop_heap*, *sort_heap* v *<algorithm>*

Brisanje iz kopice

- Recimo, da bi radi pobrisali element na indeksu *i*
  - V kopici je tam zdaj luknja – premaknimo vanjo element s konca
  - Primerjajmo ga z večjim od otrok in ga po potrebi pogreznimo (*sift*)

```
funkcija Briši(i):
    a[i] = a[n – 1]; n –= 1; Sift(i)
funkcija Sift(i): // pogrezanje
    while (i < n / 2) {  // O(log n)
        ci = 2 * i + 1;
        if (ci + 1 < n && a[ci + 1].p > a[ci].p) ci++;
        if (a[i].p <= a[ci].p) break;
        swap(a[i], a[ci]); i = ci; }
```

  - Postopek *Sift* pride prav tudi, če hočemo nekemu elementu znižati prioriteto (in moramo popraviti kopico)
  - Drobna izboljšava: namesto swap-ov si na začetku skopirajmo $a[i]$ v *temp*, nato namesto vsakega *swap* naredimo $a[i] = a[ci]$ in na koncu po zanki še $a[i] = temp$.

Dodajanje v kopico

- Novi element dodajmo na konec
  - Toda zdaj moramo preveriti, če ima mogoče višjo prioriteto od starša
  - Če jo ima, ju zamenjamo in tako naprej gor po kopici
  - To je ravno obratna operacija od dodajanja

```
funkcija Dodaj(x):
    a[n] = x; n += 1; Lift(n – 1);
funkcija Lift(i): // dvigovanje
    while (i > 0) {   // O(log n)
        starš = (i – 1) / 2;
        if (a[starš].p <= a[i].p) break;
        swap(a[starš], a[i]); i = starš; }
```

Gradnja kopice

- Recimo, da bi dano tabelo $a[0..n-1]$ preuredili v kopico
  - Vsak list je že kopica sama zase
  - Če imamo notranje vozlišče, čigar poddrevesi sta že kopici, bo s pogrezanjem starša vse skupaj postalo kopica
  - Torej poganjajmo pogrezanje od nižjih nivojev proti višjim
- Implementacija:
  funkcija *Heapify*(n):
  for ($i = (n-1) / 2$; $i >= 0$; $i--$) *Sift*(i);
- Časovna zahtevnost je le $O(n)$

Urejanje s kopico (heapsort)

- Tabelo *a*, ki bi jo uredili, najprej predelajmo v kopico
  - Največji element je zdaj v korenu – to je tisti, ki mora priti na konec tabele
  - Pobrišimo ga iz kopice
  - Na koncu s tem pridobimo v tabeli prazno mesto, prav tisto, kamor moramo vpisati pobrisani element iz korena

```
funkcija HeapSort(a[0..n – 1]):
    Heapify(a);
    while (n > 0) {
        x = a[0]; Briši(0);  // predpostavimo, da Briši zmanjša n za 1
        a[n] = x; }
```

  - Časovna zahtevnost: O(n log n)

Bisekcija

Bisekcija (binarno iskanje)

- Imamo urejeno tabelo $a[0..n-1]$, iščemo element z vrednostjo x
  - Primerjajmo x z elementom na sredi tabele
    - Če je x manjši od njega, mora biti levo, sicer desno
    - Tako na vsakem koraku zmanjšamo območje, ki ga preiskujemo, za polovico → $O(\log n)$ korakov
  - Kaj naj vrnemo, če x v tabeli ni? Ali pa če jih je več?
    - Iščimo najbolj levi element z vrednostjo vsaj x
    - Če takega ni, vrnemo n (mislimo si $a[n] = \infty$)
    - Rezultat nam bo torej tudi povedal, kam vriniti x (če ga še ni v tabeli), da bo tabela ostala urejena
  - C++: *lower_bound*, *upper_bound*, *binary_search* v <*algorithm*>

```
funkcija Bisekcija(a[0..n – 1], x):
    if (n == 0 || x <= a[0]) return 0;
    L = 0; D = n;
    while (D – L > 1) {
        // Tu velja a[L] < x ≤ a[D].
        M = (L + D) / 2;
        if (a[M] < x) L = M; else D = M; }
    // Tu velja a[L] < x ≤ a[L + 1].
    return L + 1;
```

Poizvedbe na intervalih, kumulativne vsote itd.

Poizvedbe na intervalih

- Dana je tabela $a[0..n-1]$, radi bi znali učinkovito odgovarjati na poizvedbe oblike
  - „Kakšna je vsota/minimum/maksimum elementov $a[l..d-1]$?“
  - Obstajajo razni prijemi za računanje teh stvari, odvisno od tega, ali se vsebina naše tabele lahko tudi spreminja in kako

Kumulativne vsote

- Lahko si vnaprej izračunamo tabelo kumulativnih (delnih) vsot:

  $b[i]$ = vsota vrednosti $a[0..i-1]$

  Potem je $b[d] - b[l]$ = vsota vrednosti $a[l..d-1]$

  – Porabimo $O(n)$ časa za pripravo tabele b, odtlej le $O(1)$ za vsako poizvedbo

    $b[0] = 0$; **for** $(i = 1; i < n; i{+}{+})$ $b[i] = b[i-1] + a[i-1]$;

  – Slabost: če se katerikoli element a-ja spremeni, moramo celo tabelo b izračunati znova

  – Slabost: ta prijem odpove, če nas namesto vsote zanima minimum ali maksimum

Korenska dekompozicija

https://cp-algorithms.com/data_structures/sqrt_decomposition.html

- Razdelimo v mislih tabelo a na „bloke“ po B elementov
  - Blokov je torej približno n/B
  - Za vsak blok izračunajmo vsoto (ali minimum ali karkoli nas že pač zanima)
  - Ko pride poizvedba za $a[l..d-1]$:
    - Območje $l..d-1$ pokriva na začetku in na koncu po en blok le delno, vmes pa nekaj blokov v celoti
    - Seštejmo pokrite elemente iz blokov, ki sta pokrita le delno, in vsote celih blokov, ki so pokriti v celoti
    - $O(B) + O(n/B)$ časa za eno poizvedbo
    - Najbolje bo torej pri $B = \sqrt{n} \rightarrow O(\sqrt{n})$ časa za poizvedbo
  - Če se v tabeli en element spremeni, moramo popraviti le vsoto njegovega bloka $\rightarrow O(\sqrt{n})$ časa

„Redka tabela“

https://cp-algorithms.com/data_structures/sparse-table.html

- Podobno kot ideja z bloki, le da se nam tu bloki tudi prekrivajo: $T[k, i]$ = vsota elementov $a[i..i + 2^k - 1]$
  - Tabela T porabi $O(n \log n)$ prostora, priprava tudi $O(n \log n)$ časa

    for ($i = 0$; $i < n$; i++) $T[0, i] = a[i]$;
    $k = 1$; **while** ($2^k \le n$) **for** ($i = 0$; $i < n$; i++) {
    $T[k, i] = T[k - 1, i]$;
    if ($i + 2^{k-1} < n$) $T[k, i]$ += $T[k - 1, i + 2^{k-1}]$; }
  - Ko pride poizvedba za območje $a[l..d - 1]$:
    - To območje razdelimo na bloke dolžine 2^k za razne k in skombiniramo rezultate:

      funkcija *Vsota*(l, d):
      dolžina = $d - l$; *rezultat* = 0; $k = 0$;
      while (*dolžina* > 0) {
      if ((*dolžina* & 1) == 1) { *rezultat* += $T[k, l]$; l += (1<< k); }
      dolžina >>= 1; k += 1; }
      return *rezultat*;
    - Pri minimumu je še lažje, ker lahko uporabimo dva prekrivajoča se bloka dolžine 2^k za največji tak k, pri katerem je $2^k \le d - l$: *rezultat* = min($T[k, l] + T[k, d - 2^k]$);

Drevo blokov

- Naj bo $T[k, i]$ = vsota (ali minimum itd.) območja $a[2^k i..2^k(i + 1) – 1]$
  - To si lahko predstavljamo kot zaporedje tabel: $T[k, \cdot]$ je dolga $n/2^k$ elementov
  - Lahko pa si jih predstavljamo kot drevo
  - Vse skupaj imajo $n/2 + n/4 + \ldots \approx n$ elementov
  - Če se nek $a[i]$ spremeni, moramo popraviti $O(\log n)$ elementov T-ja
- Poizvedba: na vsakem nivoju drevesa moramo pogledati največ dva elementa $\rightarrow O(\log n)$
- S še eno izboljšavo dobimo Fenwickovo drevo

Mo-jev algoritem

https://cp-algorithms.com/data_structures/sqrt_decomposition.html

- Dana je tabela $a[1..n]$ in veliko (recimo Q) poizvedb oblike (l, r) = „povej nekaj o $a[l..r]$“
  - Na primer: najpogostejši element
  - Oz. nekaj takega, za kar potrebujemo nekakšne statistike o $a[l], \ldots, a[r]$
- Recept:
  - Razdelimo tabelo na bloke velikosti B
  - Uredimo poizvedbe glede na blok, v katerem leži l
  - Za vsako tako skupino poizvedb z istim začetnim blokom:
    - Uredimo jih naraščajoče po r
    - Za prvo od njih izračunajmo odgovor z zanko od l do r
    - Za vsako naslednjo:
      - Njen r je nekje desno od r-ja prejšnje
      - Njen l je malo (največ B) levo/desno od l-ja prejšnje
      - Popravimo rezultat prejšnje poizvedbe
    - Skupaj $O(n)$ premikov r-ja in $O(B \cdot$ št. poizvedb s tem začetnim blokom) premikov l-ja
  - Po vseh začetnih blokih skupaj: $O((n/B)\, n)$ premikov r-ja in $O(B\, Q)$ premikov l-ja $\rightarrow$ pri $B = \sqrt{n}$ dobimo časovno zahtevnost $O((n + Q)\, \sqrt{n}) \cdot$ (čas popravka pri premiku l-ja ali r-ja za 1)

Primer naloge

- (UVA, #11990) Dana je permutacija števil 1..n, iz nje v nekem danem vrstnem redu brišemo elemente, po vsakem brisanju povej število inverzij – torej parov (i, j), za katere je $i < j$ in $a[i] > a[j]$
  - Ko pobrišemo element, si mislimo, da v a tam ostane luknja, tako da se indeksi ostalih elementov ne spreminjajo
  - Ko brišemo $a[i]$, nas zanima, koliko je levo od njega večjih elementov in koliko je desno od njega manjših elementov
  - Razdelimo a na bloke, za vsak blok vzdržujmo binarno iskalno drevo z vrednostmi elementov tistega bloka + števce velikosti poddreves
  - Koliko je na indeksih 0..i – 1 elementov, manjših od $a[i]$?
    - Za bloke, ki v celoti ležijo na tem intervalu, izvedemo poizvedbo v drevesu → $O(\log n)$ za vsak blok
    - Če blok leži delno na tem intervalu, pregledamo elemente posebej
  - Bloki velikosti $\sqrt{n}$: $O(\sqrt{n} \log n)$ za vsako poizvedbo, $O(\log n)$ za vsak update, $O(n \sqrt{n} \log n)$ za cel postopek
  - Bloki oblike $[2^k i..2^k(i + 1) – 1]$: $O((\log n)^2)$ za poizvedbo, enako za update, skupaj $O(n (\log n)^2)$

Primer naloge

- (Codeforces, 1175D) Zaporedje $a[1..n]$ razreži na k nepraznih delov. Naj bo $f[i] \in 1..k$ številka dela, ki mu pripada element $a[i]$. Maksimiziraj vsoto $\Sigma_{i=1..n} f[i]\ a[i]$.
  - Naj bo p_j indeks prvega elementa v j-tem delu
  - Našo vsoto lahko zapišemo tudi kot
    $$\Sigma_{j=1..k} \Sigma_{i=p_j..n} a[i] = \Sigma_{j=1..k} s[p_j],$$
    če je $s[i] = a[i] + \ldots + a[n]$
  - Vidimo torej, da moramo le izbrati največjih $k - 1$ izmed delnih vsot $s[2..n]$ in jih sešteti (in prišteti še $s[1]$)
    - Quickselect, $O(n)$

Nizi in iskanje podnizov

Nizi

- Niz (*string*) je zaporedje znakov
  - $|s|$ je dolžina niza
  - Če sta s in t niza, je st stik teh dveh nizov
  - Znake označimo z $s[0], s[1], \ldots, s[n-1]$ za $n = |s|$
  - Podnizi: $s[i:j] = s[i]\ s[i+1]\ \ldots\ s[j-1]$
  - $s[0:i]$ = prefiks (začetek, predpona)
  - $s[i:n]$ = sufiks (konec, končnica, pripona)

Iskanje podniza v nizu

- Dana sta niza *s* in *p*; ali se (oz. kje se) pojavlja *p* kot (strnjen) podniz *s*-ja?
  - Pišimo $n = |s|$ in $m = |p|$
- Naivna rešitev: $O(nm)$

```
funkcija NajdiPodniz(s, p):
    for (i = 0; i < n – m; i++) {
        j = 0; while (j < m && s[i + j] == p[j])  j++;
        if (j == m) return i ; }
    return –1; }
```

  - V povprečju sicer ne bo tako slaba, če se neujemanje opazi hitro (npr. če sta *s* in *p* iz naključnih znakov)

Rabin-Karpov algoritem

- Ideja: preden pri nekem *i* primerjamo *s*[*i*:*i*+*m*] in *p* znak po znak, izračunajmo iz vsakega od njiju neko hash kodo
  - Če se hash kodi ne ujemata, sta niza gotovo različna in ju ni treba primerjati

```
funkcija RabinKarp(s, p):
    hp = hash(p);
    for (i = 0; i < n – m; i++) {
        hs = hash(s[i:i + m]); if (hs != hp) continue;
        j = 0; while (j < m && s[i + j] == p[j])  j++;
        if (j == m) return i ; }
    return –1; }
```

  - Večinoma bosta hash kodi enaki le, če bosta tudi niza enaka, tako da bo primerjanje po znakih potrebno le enkrat → $O(n + m)$
  - …če znamo te hash kode poceni računati (ne pa s še eno zanko po znakih niza *s*[*i*:*i* + *m*])

Rolling hash kode

- Znake niza lahko gledamo kot cela števila (npr. po ASCII ipd.) – kot števke v B-iškem sestavu za nek velik B
  - Niz k znakov je kot zaporedje k števk – torej kot neko (ogromno) celo število
  - Za hash kodo vzemimo njegov ostanek po deljenju z M:
    $hash(t[0{:}k]) = (\Sigma_{i=0..k-1}\ B^{k-1-i}\ t[i])\ \%\ M$
  - Lahko ga računamo z zanko:

    funkcija $hash(t[0{:}k])$:
    $h = 0$; **for** $(i = 0; i < k; i{+}{+})\ h = (h * B + t[i])\ \%\ M$;
    return M;
  - Ko se v glavni zanki premaknemo naprej po nizu s:
    $hash(s[i{+}1{:}i{+}1{+}m]) = ((hash(s[i{:}i{+}m]) - B^{k-1}\ s[i]) * B + s[i+m])\ \%\ M$
    - Torej lahko novo hash kodo izračunamo iz stare v $O(1)$ časa, če si vnaprej pripravimo $B^{m-1}\ \%\ M$

Z-funkcija

https://cp-algorithms.com/string/z-function.html

- Za dani niz $s[0{:}n]$ definirajmo tabelo $z[i]$ = dolžina najdaljšega skupnega prefiksa nizov $s[0{:}n]$ in $s[i{:}n]$
  - Da poiščemo pojavitve podniza p v nizu s, izračunamo tabelo z za niz $p\#s$ in v njej iščemo indekse, ki imajo $z[i] = |p|$
  - Vprašanje je, kako poceni računati tabelo z, naivna rešitev je v $O(n^2)$

    for ($i = 0$; $i < n$; i++) {
    $z[i] = 0$; **while** ($i + z[i] < n$ && $s[i] == s[i + z[i]]$) $z[i]$++; }
  - Vsak doslej izračunani $z[i]$ nam pove, da je $s[0{:}z[i]] = s[i{:}i + z[i]]$.
    Med njimi si zapomnimo tistega, ki seže najbolj desno, torej z največjim $i + z[i]$.
    Temu recimo $s[l{:}d]$.
  - Ko kasneje računamo $z[i]$ za nek večji i ($i > l$):
    - Če je $i > d$, računamo kot pri naivni rešitvi
    - Sicer začnemo pri $z[i] = \min\{z[i - l], d - i\}$ in nadaljujemo kot pri naivni rešitvi
    - Zakaj? Ker je $s[l{:}d] = s[0{:}d - l]$, je $s[i{:}d] = s[i - l{:}d - l]$
    - Za $s[i - l{:}n]$ že vemo, da se ujema s $s[0{:}n]$ v prvih $z[i - l]$ znakih
    - Torej se $s[i - l{:}\, d - l]$ ujema s $s[0{:}n]$ v prvih $\min\{z[i - l], d - i\}$ znakih

Knuth-Morris-Prattov algoritem

- Recimo, da smo primerjali $s[0:m]$ in p ter opazili prvo neujemanje po j znakih
- Če zdaj premaknemo p še i (za nek $i < j$) mest naprej…
  - Se mora $p[0:]$ ujemati s $p[i:]$ v vsaj prvih $j - i$ znakih (*)
  - Prvo neujemanje s s-jem gotovo ne nastopi prej kot po $j - i$ znakih
- p-ja torej ni treba zamikati po 1 znak naprej, vzamemo lahko najmanjši premik, pri katerem je izpolnjen pogoj (*)
  - Primerjanje z znaki s-ja pa nadaljujemo pri $s[j]$, kjer smo ga prej končali
- Vsak znak s-ja sodeluje v kvečjemu eni uspešni primerjavi, za vsak položaj p-ja je še kvečjemu ena neuspešna primerjava → $O(n + m)$
- Vnaprej si za vsak j izračunamo najmanjši primerni zamik i
  - Torej najmanjši i, za katerega je $p[0:j - i] = p[i:j]$
  - Naivno v času $O(m^3)$, dá se tudi v $O(m)$

Boyer-Mooreov algoritem

- Recimo, da smo pri nekem i in da primerjamo $p[0{:}m]$ in $s[i{:}i+m]$
  - Gremo od desne proti levi
  - Ko opazimo neujemanje, za koliko naj povečamo i?
    - Recimo, da sta se $p[0{:}m]$ in $s[i{:}i+m]$ ujemala v zadnjih k znakih, pred tem pa je $p[m-k-1] \neq s[i+m-k-1] = c$
  - i moramo povečati vsaj za toliko, da bo pod $s[i+m-k{:}i+m]$ stala predzadnja pojavitev tega niza v p-ju (zadnja je čisto na koncu p-ja)
  - i moramo povečati vsaj za toliko, da bo pod $s[i+m-k-1]$ stala zadnja pojavitev c-ja v nizu $p[0{:}m-k]$
  - Oboje imamo potabelirano vnaprej [$O(m)$], uporabimo večjega od obeh premikov
  - C++17: *boyer_moore_searcher* v headerju <*functional*>

Primer naloge

- (UVA, #11475) Dan je niz s, dodaj na koncu nek čim krajši niz t tako, da bo st palindrom
  - Palindrom bo torej oblike $st = t^R\ u^R\ c\ u\ t$ za $s = t^R\ u^R\ c\ u$
  - Iščemo torej nek čim daljši konec (sufiks) s-ja, u, ki se malo pred tem pojavi kot u^R
  - Ali pa: iščemo čim daljši w, ki se pojavi kot w^R tudi na koncu s-ja
  - Izračunajmo Z-tabelo nad nizom $s^R\#s$
  - V desni polovici te tabele nam $z[n + 1 + i] = d$ pove, da se $s[i:]$ ujema s s^R v prvih d znakih
  - Če se tidve pojavitvi stakneta ($i + d \geq n - d$), tvorita primeren $u^R\ c\ u$ in za t lahko vzamemo $s[:i]$

Domače naloge

- https://codeforces.com/problemset/problem/1005/E1
  - Dana je tabela a, ki vsebuje permutacijo števil $1..n$
  - Za dani m povej, koliko je parov (l, r), za katere velja, da je mediana elementov $a[l..r]$ enaka m
- https://codeforces.com/problemset/problem/58/D
  - Danih je $2n$ nizov in ločilni znak #
  - Zloži po dva niza v vrstico oblike $s\#t$
  - Nastane seznam n vrstic, hočemo leksikografsko najmanjšega
- http://codeforces.com/problemset/problem/86/D
  - Dana je tabela $a[1..n]$, odgovarjaj na poizvedbe oblike
    $$Q(l, r) = \Sigma_x N(l, r, x)^2 \, x,$$
    kjer je $N(l, r, x)$ število pojavitev vrednosti x v $a[l..r]$
  - Namig: Mo-jev algoritem