**STRUKTURE PODATAKA**

**I ALGORITMI 2**

ODGOVORI NA PITANJA ZA

USMENI DEO ISPITA

JANUAR 2021

Git: <https://github.com/NikolaVetnic/SPA2>

Sortiranje niza grubom silom i umetanjem.

01

**SORTIRANJE NIZOVA:** sortiranje po prirodnom uređenju koje postoji kada objekti implementiraju *Comparable* interfejs, sortiranje po eksternom kriterijumu koji je definisan komparatorom.

**SORTIRANJE GRUBOM SILOM:** dva elementa a[i] i a[j] su u inverziji ako je i < j i a[i] > a[j], pa se sortiranje grubom silom (brute force) svodi na poređenje svaka dva elementa ponaosob i njihovu razmenu ukoliko su u inverziji. Ukupan broj poređenja za elemenata je n elemenata je n(n – 1)/2 pa je vremenska složenost O(n2) – u najgorem slučaju operacija je kvadratne složenosti veličine ulaza.

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p01_elementary_sorts>

**SORTIRANJE UMETANJEM:** ideja je da se niz sastoji od sortiranog (sa leve strane) i nesortiranog dela, gde se prvi element iz nesortiranog ubacuje na odgovarajuće mesto u sortiranom (čime se sortirani uvećava za element), što se ponavlja dok se ne sortira čitav niz; početak je od prvog elementa koji predstavlja sortiran, a ostatak nesortiran deo. Postupak: neka je deo niza [0, i-1] sortiran – u svakom koraku se povećava za jedan, pa se i kreće u [1, arr.length – 1]; prvi nesortirani element je arr[i]; ukoliko je arr[i] > arr[i-1] ne radimo ništa budući da je on zapravo već sortiran, u suprotnom tražimo indeks j < i takav da je arr[j] ≤ arr[i] – arr[i] tada dolazi na poziciju j+1, elemente od arr[j+1] do arr[i-1] pomeramo za jedno mesto nadesno (što se vrši dok tražimo indeks j), pri čemu čuvamo referencu na arr[i] (jer će ga u pomeranju prebrisati arr[i-1]).

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p01_elementary_sorts>

Sortiranje niza izabiranjem i razmenom.

02

**SORTIRANJE IZABIRANJEM:** niz je sastavljen iz dva dela – sortiranog (sa desne strane) i nesortiranog, čime su svi elementi iz nesortiranog manji od svih iz sortiranog dela niza; traži se maksimalni element nesortiranog dela postavlja se na njegov kraj. Druga varijanta sortiranja deli niz na sortirani deo sa leve i nesortirani sa desne strane; traži se minimum desne strane i postavlja na početak nesortiranog dela niza.

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p01_elementary_sorts>

**STABILNA I NESTABILNA SORTIRANJA:** postupak sortiranja je stabilan ako za svako i i j, takvo da je a[i] = a[j] u polaznom nizu, element a[i] u sortiranom nizu bude pre a[j] – relativni redosled identičnih elemenata biva očuvan kod stabilnog sortiranja. Po *default*-u sortiranje umetanjem je stabilno a izabiranjem nije (iako se može napraviti da bude). Stabilna sortiranja dozvoljavaju sortiranje po dva kriterijuma, i to prvo po sekundarnom, a onda po primarnom – sortiranje po sekundarnom ostaje očuvano zahvaljujući stabilnosti sortiranja.

**SORTIRANJE RAZMENOM:** zasniva se na ideji da će, ako svaka dva susedna elementa koja su u inverziji zamene mesta, maksimalni element „isplivati“ na kraj niza (kraj nesortiranog dela varira od n – 1 kada je nesortirani deo čitav niz, do 1 kada nesortirani deo niza čine dva elementa); inverzni postupak ima analogni efekat na minimalni element (početak nesortiranog dela varira od 0 do n – 2). Sortiranje se može unaprediti tako što se sortiranje prekine posle iteracije u kojoj nije bilo razmene elemenata.

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p01_elementary_sorts>

Sortiranje niza *Shell sort*-om.

03

**SHELL SORT:** (*Donald Shell*, 1959) osnovna ideja – 1) ukupan broj parova u inverziji se može posmatrati kao mera neuređenosti niza, 2) kod elementarnih sortiranja pri svakom poređenju susednih elemenata ukupan broj inverzija se smanjuje za jedan, 3) razmena nesusednih elemenata u nizu može smanjiti više od jedne inverzije. Niz A dužine n se deli na podnizove A0, A1… Ak-1 za neko k na sledeći način: A0 = (0, k, 2k…), A1 = (1, k+1, 2k+1…) … Ak-1 = (k-1, 2k-1…) – drugim rečima, podniz Aj sadrži indekse koji pri deljenju sa k daju ostatak j; podnizovi se sortiraju umetanjem; postupak se ponavlja za vrednosti k koje se smanjuju od neke početne vrednosti do 1 sa nekim korakom; kada je k = 1 dešava se sortiranje umetanjem nad celim nizom.

Kako varirati k – 1) sam *Shell* predložio je vrednosti , … … 1, gde je n veličina niza (*Shell sort* se značajno ubrzava kada k varira tako da bude stepen broja 2 uvećan ili umanjen za 1), 2) (*Tokuda*, 1992) sortiranje daje najbolje performanse kada se k sukcesivno smanjuje 2.25 puta; 3) (*Ciura*, 2001) sortiranje daje najbolje performanse kada k uzima redom vrednosti { 701, 301, 132, 57, 23, 10, 4, 1 }.

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p01_elementary_sorts>

Sortiranje niza *Comb sort*-om.

04

**COMB SORT:** zasniva se na razmeni nesusednih elemenata u nizu, predstavlja unapređenje *Bubble sort*-a (sortiranje razmenom); kod sortiranja razmenom veći elementi „brzo prilaze“ kraju niza, dok manji „sporo prilaze“ početku niza – osnovna ideja je eliminisati „efekat kornjača“ („sporih“ elemenata) razmenom nesusednih elemenata u nizu.

**Algoritam**: 1) za svaki element proveravamo da li je u inverziji sa elementom udaljenim k pozicja udesno i ako jeste razmenimo ih, 2) ponavljamo *Comb* prolaz smanjujući k, 3) niz je sortiran kada je k = 1 i nije bilo razmena elemenata. Drugim rečima, *Comb* prolaz se za k = 1 ponavlja sve dok postoje elementi u inverziji što je zapravo *Bubble sort*.

Kako varirati k – eksperimentalno je utvrđeno da *Comb sort* daje najbolje performanse kada se k sukcesivno smanjuje 1.3 puta, ali se preskoče k = 9 i k = 10 i uzme se k = 11.

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p01_elementary_sorts>

Sortiranje niza *Heap sort*-om.

05

**HEAP SORT:** unapređenje sortiranja izabiranjem linearitamske složenosti (O(n logn)).

**Algoritam**: 1) neka je A niz dužine n, 2) transformišemo A tako da ime strukturu *heap*-a (maksimum je tada prvi u nizu), 3) razmenimo maksimum sa elementom na kraju niza, 4) povratimo strukturu *heap*-a na podnizu A[0 .. n-1] i maksimum zamenimo sa elementom na poz. n – 2, 5) ponavljamo postupak do podniza A[0 .. 1] kada maksimum zamenimo sa susednim elementom.

***Heap* osobina**: niz ima *heap* osobinu ukoliko je svaki element veći ili jednak svojim sinovima, gde sinovi roditelja indeksa p imaju indekse 2p + 1 i 2p + 2, a roditelj sina indeksa s ima indeks (s – 1)/2 (celobrojno deljenje); kod niza dužine n elementi sa idx u opsegu [0, (n – 2)/2] su očevi. Počevši od poslednjeg ka prvom ocu uspostavljamo *heap* strukturu tako što: 1) proverimo da li je otac manji od većeg sina, 2) ako jeste, zamenimo oca sa većim sinom, 3) ponavljamo **1)** i **2)** dokle god otac ne postane veći od sinova.

<https://github.com/NikolaVetnic/SPA2/blob/master/src/p03_advanced_sorts/>

Sortiranje niza *Quick sort*-om.

06

**QUICK SORT:** (*Tony Hoare*, 1959) jedan od najčešće korišćenih u praksi, u proseku linearitamske vremenske složenosti (O(n logn)), zasniva se na *divide and conquer* principu. Ideja: 1) odaberemo jedan element koji nazovemo pivot, 2) preuredimo niz tako da je oblika (LE pivot QE), gde su LE elementi manji od ili jednaki pivotu, a QE veći ili jednaki pivotu (postupak particionisanja), 3) sortiramo LE i QE *quick sort*-om (najčešće realizovano rekurzivno). Particionisanje niza: selekcija pivota i transformacija u formu (LE pivot QE), za čega postoji više šema (*Hoarova* – pivot je A[l], *Lomutoova* – pivot je A[h], i ona gde je pivot na sredini niza). Opšti oblik za prve dve šeme:

public static <T extends Comparable<T>> void sort(T[] arr) {

sort(arr, 0, arr.length – 1); }

private static <T extends Comparable<T>> void sort(T[] arr, int l, int h) {

int j = partition(arr, l, h);

sort(arr, l, j – 1);

sort(arr, j + 1, h); } }

**Hoarova šema**: za pivot se uzima prvi el. niza (idx l), kreiraju se idx i (na l+1 poziciji) i j (na kraju niza); idx i se kreće nadesno preskačući sve el. strogo manje od pivota dok ne dođe do el. x koji ne ispunjava taj uslov – analogno idx j preskače stroko veće od pivota dok ne dođe do el. y, kada se dešava zamena x i y, nakon čega se kretanje idx i, pa j ponavlja; kada se idx i i j mimoiđu (j<i) dolazi do zamene pivota i el. sa idx j jer on pripada delu koji je manji od pivota. Demo: <https://www.youtube.com/watch?v=cnzIChso3cc>

**Lomutova šema**: za pivot se uzima poslednji el. niza (idx h) i sekvencijalno se prolazi kroz čitav niz od idx l do h-1 i tom prilikom se niz deli na dva dela – sa el. manjim ili jednakim pivotu i el. strogo većim od pivota; svaki sledeći element (idx i) može biti: 1) manji ili jednak pivotu – tada se korišćenjem posebnog idx ltePivot on menja sa prvim elementom dela niza većeg od pivota čime se deo manji ili jednak pivotu uvećava za jedan, i 2) veći od pivota – tada se ne radi ništa; finalni korak je razmena pivota sa idx ltePivot. Demo: <https://www.youtube.com/watch?v=86WSheyr8cM>

**Sredina niza**: idx i kreće od l i preskače strogo manje, idx j od h i preskače strogo veće, pivot je na (i+j)/2; kada se zaustave idx i i j dolazi do zamene; postupak se ponavlja dok se idx i i j ne mimoiđu, s tim da ovde pivot ne ostaje na mestu već završi u delu niza suprotnom od idx koji je pre stigao do njega (desnom ako je prvi i, npr.); slede dva rekurzivna poziva za opsege [l,j] i [i,h]. Nema provere izlaska iz opsega niza jer se pivot ponaša kao graničnik za onaj idx koji prvi stigne do njega. Demo: <https://www.youtube.com/watch?v=Hoixgm4-P4M>

<https://github.com/NikolaVetnic/SPA2/blob/master/src/p03_advanced_sorts/>

Sortiranje liste umetanjem.

07

**SORTIRANJE UMETANJEM:** svaki postupak za sortiranje niza može se adaptirati za sortiranje listi, ali neki postupci su prirodniji listama (npr. baš sortiranje umetanjem je prirodno listama jer se element trivijalno umeće na drugo mesto u listi, dok se kod niza moraju pomerati elementi).

Ako je lista prazna ili jednoelementna ona je sortirana. Neka je levi deo sortiran a desni nesortiran, tada: lastSorted je pokazivač na poslednji element u sortiranom, a firstUnsorted na prvi u nesortiranom delu (pa je firstUnsorted = lastSorted.next); razlikuju se tri slučaja: 1) firstUnsorted >= lastSorted – nema umetanja, sortirani deo se trivijalno povećava (lastSorted = firstUnsorted); 2) firstUnsorted < root – firstUnsorted se umeće pre root-a; 3) inače – firstUnsorted treba umetnuti iza lastLeq (lastLeq pokazuje na poslednji element između root i firstUnsorted koji je manji ili jednak od firstUnsorted, što se lako pronalazi pretraživanjem liste od korena ka kraju).

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p04_sortable_list>

Sortiranje liste *Merge* sortiranjem.

08

**MERGE SORT:** zasniva se na *divide and conquer* principu, linearitamske složenosti (O(n logn)). Osnovna ideja: 1) podelimo listu L u dve balansirane liste L1 i L2 (dužine L1 i L2 se razlikuju najviše za 1), 2) sortiramo L1 i L2 *merge sort*-om, 3) spojimo dve sortirane liste L1 i L2 u finalnu sortiranu listu L. Sortiranje se tako sastoji od dva potproblema: 1) podela liste u dve balansirane liste, i 2) spajanje sortiranih listi u sortiranu listu.

**Podela liste**: neka start pokazuje na koren originalne liste koja ima bar dva elementa, l1 i l2 pokazuju na korene L1 i L2, l1End i l2End pokazuju na poslednje elemente L1 i L2; podela počinje inicijalizacijom pokazivača:

l1 = start; l1End = l1; l2 = start.next; l2End = l2;

Tako l1 i l1End pokazuju na koren, a l2 i l2End na drugi element originalne liste; nakon inicijalizacije ažuriraju se samo l1End i l2End (elementi iz originalne liste se dodaju na kraj L1 ili L2); elementi se dodaju u L1 i L2 naizmenično, i to tako što se dodaju na kraj liste, nakon čega se povrati esencijalno svojstvo current pokazivača.

Na kraju, voditi računa o tome da element koji ide u L1 može biti poslednji (kod liste sa neparnim brojem elemenata) pa se pre dodavanja u L2 mora raditi provera na null; takođe nakon deljenja potrebno je „uzemljiti“ L1 i L2: l1End.next = null; l2End.next = null;

**Spajanje sortiranih listi**: neka root pokazuje na koren liste koja se dobija spajanjem dve sortirane liste L1 i L2, last pokazuje na poslednji element u spojenoj listi, l1 i l2 na prvi element koji nije dodat u spojenu iz L1 i L2 (na početku l1 i l2 pokazuju na korene L1 i L2). Spajanje se realizuje u tri koraka:

1) određivanje korena spojene liste – ako je prvi el. L1 manji od prvog el. L2 tada (analogno u obrnutom slučaju):

root = l1; l1 = l1.next; (povraćaj esencijalne osobine l1)

last = root; (prvi el. spoljne liste je na početku i njen poslednji)

2) dodavanje elemenata iz L1 i L2 u spojenu listu – razlikujemo slučaj kada je l1.info < l2.info i tada dodajemo l1 na kraj spojene liste (analogno u obrnutom slučaju) – ova operacija se ponavlja dokle god postoje elementi u obe liste:

last.next = l1;

last = l1; (povraćaj esencijalne osobine last)

l1 = l1.next; (povraćaj esencijalne osobine l1)

3) kalemljenje ostatka kada se jedna od listi „isprazni“ – vrši se jednostavnom proverom i prevezivanjem pokazivača:

last.next = l1 == null ? l2 : l1;

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p04_sortable_list>

Sortiranje liste *Quick* sortiranjem.

09

**QUICK SORT:** bazira se na ideju pivota i podele liste na dva dela spram pivota; neka je L = (G|R) lista koja se sortira, gde je G pivot (za koji selektujemo el. koji se najbrže i najlakše dobavlja – dakle koren) a R ostatak liste koju delimo na dva dela: M (manji od pivota) i V (veći od pivota); sortiramo M i V i dobijamo M’ i V’, pa je tada sortirana lista L’ = (M’|G|V’). Formiranje lista M i V se realizuje obilaskom liste G. Budući da sortirana lista L’ ima oblik L’ = (M’|G|V’) lista M’ može biti prazna lista – tada je koren L’ liste čvor G a ne prvi el. iz M’; ako pak M’ nije prazna, tada G treba nakalemiti na kraj M’ (šetnja do kraja liste M’); kalemljenje V’ na G je trivijalno.

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p04_sortable_list>

ADT prioritetna lista, oepracije *insert*, *max* i *isEmpty*.

10

**TEXT:** text