**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 n elemenata je n(n – 1)/2 pa je vremenska složenost O(n2) – u najgorem slučaju operacija je kvadratne složenosti veličine ulaza.

**SORTIRANJE UMETANJEM (INSERTION):** 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. Postupak: 1) početak je od prvog elementa koji predstavlja sortiran, a ostatak nesortiran deo; 2) neka je deo niza [0, i-1] sortiran – u svakom koraku se povećava za jedan, pa se i kreće u [1, arr.length – 1]; 3) prvi nesortirani element je arr[i]; 4) 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]). Optimizacija: kod jezika koji ne podržavaju lenje računanje logičkih izraza potrebno je pre sortiranja pronaći minimalni element i staviti ga na početak.

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qA01_bruteforce_insertion>

Sortiranje niza izabiranjem i razmenom.

02

**SORTIRANJE IZABIRANJEM (SELECTION):** 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.

**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 (BUBBLE):** 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). Optimizacija: 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/p_vezba/qA02_selection_bubble>

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 }.

Demo: <https://www.youtube.com/watch?v=SHcPqUe2GZM>

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qA03_shell>

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 ili nije bilo razmena elemenata[[1]](#footnote-1). 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.

Demo: <https://www.youtube.com/watch?v=n51GFZHXlYY>

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qA04_comb>

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 – 1)/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.

**public** **static** **<T..** **>** **void** **sort(T[] arr)**: uzima se idx poslednjeg elementa i poslednjeg oca i u while petlji se uspostavlja *heap*; int end dobija vrednost kraja niza i ulazi kao kontrola while petlje tokom koje koren i poslednji element menjaju mesta, end se dekrementira i ponovo se uspostavlja *heap*.

**private** **static** **<T..** **>** **void** **makeHeap(T[] arr, int start, int end)**: int parentIndex uzima start, uvodi se flag heapRestored; ulaz u while petlju sa uslovom !heapRestored, računa se maxSonIndex za parentIndex i proverava se da li je == -1 – ako da petlja se prekida, inače sledi provera da li je element sa idx oca manji od onog sa idx sina – ako da zamenjuju mesta i parentIndex uzima vrednost sina (dakle, parentIndex ide ka većoj vrednosti), inače se petlja prekida.

**private** **static** **<T..** **>** **int** **getMaxSon(T[] arr, int parentIndex, int end)**: računaju se dva moguća idx za datog oca; provera da li je prvi sin van opsega niza, ako nije maxSonIndex uzima njegovu vrednost; provera da li je drugi sin van opsega niza, ako nije provera da li je njegov element veći od elementa sa maxSonIndex, ako jeste maxSonIndex uzima njegovu vrednost; vraća se maxSonIndex (čija *default* vrednost je -1).

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qA05_heap>

Sortiranje niza *Quick sort*-om (opis i implementacija Lomutove šeme particionisanja i bar još jedne po izboru).

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 element niza (idx l), kreiraju se idx i (na l+1 poziciji) i j (na kraju niza); idx i se kreće nadesno preskačući sve elemente strogo manje od pivota dok ne dođe do elementa x koji ne ispunjava taj uslov – analogno idx j preskače strogo veće od pivota dok ne dođe do elementa 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>

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qA06_quick_hoare>

***Lomutova* šema**: za pivot se uzima poslednji element 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 elementima manjim ili jednakim pivotu i elementima 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>

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qA06_quick_lomuto>

**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/tree/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 (deklaracija i inicijalizacija VAN while petlje), 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/p_vezba/qA07_list_insertion>

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 (jednu) 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 element L1 manji od prvog elementa 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/p_vezba/qA08_list_merge>

Sortiranje liste *Quick* sortiranjem.

09

**QUICK SORT:** bazira se na ideji 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 element koji se najbrže i najlakše dobavlja – koren) a R ostatak liste koju delimo na dva dela: M (manji od G) i V (veći od G); 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/p_vezba/qA09_list_quick>

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

10

**PRIORITETNA LISTA:** kolekcija objekata u kojoj svaki element ima prioritet obrade (ideja uvođenja prioriteta tako da se podaci sa većim prioritetom ranije procesiraju), pri čemu se element sa najvećim prioritetom procesira prvi; elementi nisu nužno sortirani po prioritetu, bitno je samo da se u svakom trenutku zna koji ima najveći prioritet.

**Prioritet**: 1) eksplicitni – svakom elementu se pridružuje ceo/realan broj kojim je određen prioritet, i 2) implicitni – prioritet određen prirodnim uređenjem (objekti klase koja implements Comparable, MAX-PQ i MIN-PQ) ili proizvoljnim komparatorom.

Tip podataka ADT-a određuje *skup vrednosti i operacija* nad varijablama tog tipa – način realizacije ADT-a *sakriven od korisnika tipa* (korisnik ne mora da zna detalje implementacije kako bi koristio tip). Kod ADT prioritetne liste **vrednost tipa** je kolekcija uporedivih elemenata (može biti struktuirana na više fundamentalno različitih načina), a **operacije** su: 1) dodavanje novog elementa, 2) dobavljanje elementa sa najvećim prioritetom, 3) brisanje elementa sa najvećim prioritetom, 4) provera da li je lista prazna, 5) veličina liste.

**Specifikacija ADT-a**: apstraktni deo ADT-a; zaglavlja public metoda kojima se definišu operacije ADT-a, to je ono što korisnik treba da zna da bi koristio tip; različite implementacije ADT-a dele istu specifikaciju. U Java-i se ADT može specifirati interfejsima:

public interface PriorityQueue<T extends Comparable<T>> {

void insert(T element); T max();

T delMax(); boolean isEmpty();

int size(); }

**Reprezentacija prioritetne liste**: izvodi se nizom koji ima *heap* osobinu, što obezbeđuje efikasne *insert/delete* operacije logaritamske složenosti (O(logn)); ovakav niz se može predstaviti u obliku binarnog stabla, što je struktura koja se sastoji od čvorova pri čemu svaki čvor ima pokazivače na levo i desno podstablo. Terminologija: 1) svaki čvor ima najviše dva sina; 2) koren nije nikome sin, a listovi nemaju sinove; 3) otac čvora B je A ako je B sin od A.

**BINARNO STABLO SA *HEAP* OSOBINOM:** zadovoljava sledeće uslove: 1) stablo je kompletno (svi nivoi osim možda poslednjeg su potpuno popunjeni, a poslednji ako nije potpun je popunjen redom sleva nadesno; 2) za svaki čvor važi da je veći od svojih sinova (odnosi se na oba podstabla, koren je maksimum). Ovakva stabla jednostavno se predstavljaju nizom, gde su sinovi sa indeksima 2p + 1 i 2p + 2, a roditelj čvora sa indeksom s je (s – 1) / 2 (celobrojno deljenje).

**void insert(T element)**: 1) dodaje se novi element na kraj (proširivog) niza; 2) popravlja se niz tako da se ponovo uspostavi *heap* osobina preko indeksa dodatog sina.

***void restoreHeapProperty(int sonIndex)***: računa se indeks oca i ako je novi element veći od njega razmenjuju se, što se ponavlja sve dok je zadovoljen uslov razmene ili dok se ne stigne do korenskog čvora (dakle, sonIndex ide naniže).

**T max()**: proverava se da li je lista prazna i ako nije vraća se koren liste (idx = 0).

**boolean isEmpty()**: proverava da li je veličina liste jednaka 0.

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qA10_11_priority_queue>

ADT prioritetna lista, *delMax* i *size*.

11

**T delMax()**: 1) poslednji element niza zameni mesta sa korenom; 2) povrati se *heap* struktura metodom void restoreHeapProperty() – bez argumenata!

***void restoreHeapProperty()***: kretanjem od korena ka kraju niza pronalazi se idx većeg od dva sina metodom int getMaxSon(int parentIndex); ukoliko takav ne postoji trenutni element je list i *heap* osobina je uspostavljena, ukoliko postoji upoređuju se otac i (veći) sin – ukoliko je otac veći *heap* osobina je uspostavljena, ukoliko nije otac i (veći) sin zamene mesta, sin postaje otac i postupak se ponavlja (dakle parentIndex ide od 0 naviše).

***int getMaxSon(int parentIndex)***: računaju se idx oba (potencijalna) sina, izlazna promenljiva se inicijalizuje na -1, proverava se da li postoji element sa idx prvog, a zatim i drugog sina – ako ne postoji izlazna promenljiva dobija vrednost prvog sina, a ako postoji upoređuju se sinovi i izlazna dobija vrednost većeg.

**int size()**: vraća se veličina niza.

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qA10_11_priority_queue>

ADT skup realizovan otvorenih hešovanjem.

12

***HASH* TABELA**: niz u kom se pozicija elementa određuje na osnovu njega samog pomoću *hash* f-je koja izračunava poziciju: h : Object → [0 .. M-1], gde je M veličina hash tabele; *hash* f-ja mora da bude brzo i lako izračunljiva. *Hash* tabelama realizuju se dva bitna ADT-a: skupovi i mape, budući da je pretraživanje znatno brže u odnosu na realizacije nizovima i listama.

**ADT SKUP**: kolekcija objekata bez duplikata. Operacije: 1) dodavanje elementa, 2) brisanje elementa, 3) pretraživanje – da li je element u skupu, 4) unija, presek, razlika (često su dovoljne samo prve tri, uz napomenu da i dodavanje i brisanje zahtevaju pretraživanje).

**ADT MAPA**: mapa, iliti asocijativni niz / tabela simbola / rečnik, je kolekcija objekata oblika (ključ, vrednost) pri čemu svaki objekat ima jedinstven ključ (nema duplikata ključeva). Operacije: 1) dodavanje (k, v) para u mapu – provera da li već postoji k u mapi i nastavljamo ako ne; 2) brisanje (k, v) para iz mape; 3) pretraživanje (*lookup*) – dobavljanje vrednosti za dati ključ (null ukoliko ne sadrži k), najbitnija operacija; 4) modifikovanje v za dati k.

***HASH* KOD**: *hash* f-ja vraća vrednost – *hash* kod – jedinstvenu za objekat, i ona mora biti konzistentna (k1 = k2 → hash(k1) = hash(k2)); metod hashCode() definisan u klasi Object koristi se da se kreira *hash* f-ja u opštem slučaju. *Hash* f-ja vraća broj u opsegu [0 .. M-1] za veličinu tabele M (veza između *hash*-a i *hash* koda):

public int hash(Object o, int hashTableSize) {

return Math.abs(o.hashCode() % hashTableSize); }

Redefinišući metod hashCode() u našoj klasi dobijamo *hash* kod za objekte te klase, a oni se mogu proslediti u gornju metodu hash() kako bismo dobili vrednosti *hash* f-je za naše objekte. **Kolizije**: dva objekta mogu imati istu vrednost *hash* f-je i to se naziva kolizijom (verovatnoća kolizije je neintuitivan (nelinearan) fenomen, veća je nego što intuicija sugeriše)[[2]](#footnote-2). **Princip uniformnosti**: neka je M veličina *hash* tabele i О skup objekata gde je |O|>>M, tada *hash* f-ja particioniše О u familiju skupova Hk, 0≤k<М, gde Hk sadrži objekte sa istom vrednošću *hash* f-je; dobra *hash* f-ja: |H0| ≈|H1| ≈ ... ≈|HM-1| ≈ |O|/M

**OTVORENO HEŠOVANJE (ZATVORENO ADRESIRANJE)**: jedan od mehanizama razrešenja kolizija, *hash* tabela je niz JPL, liste (lanci kolizija) sadrže objekte sa istom vrednošću *hash* f-je; za veličinu *hash* tabele bira se neki veliki prost broj. Demo: <https://tinyurl.com/g4g-hashSet>

**Vrednosti**: da bi se napravio skup objekata klase A koristeći OHashSet klasa A mora da redefiniše sledeće metode klase Object: int hashCode(), boolean equals(Object o), String toString().

**private static class Node { … }**: *wrapper* klasa za realizaciju lanaca kolizije, sadrži Object value i Node next pokazivače.

**OHashSet()**, **OHashSet(size)**, **hash(T o)**: konstruktori i hash f-ja (hashCode() % table.length).

**Node[] searchCollisionChain(T element, int hashValue)**: krećući se od korena lanca kolizije proverava za svaki element u lancu da li je jednak traženom i ako jeste vraća niz od dva pokazivača, i to na traženi element i onaj koji mu prethodi.

**boolean member(T element)**: pretraga lanca kolizije za *hash* vrednost elementa.

**boolean insert(T element)**: pretraga lanca kolizije za *hash* vrednost elementa, ukoliko postoji vraća false, ukoliko ne postoji element se stavlja u Node i prevezuje na početak lanca.

**boolean remove(T element)**: pretraga lanca kolizije za *hash* vrednost elementa, ukoliko ne postoji vraća false, ukoliko postoji pita se da li je to koren lanca i ako jeste izvršava table[hashValue]=table[hashValue].next, a ako nije n[1].next=n[0].next (n je nizovni rezultat pretrage lanca kolizije).

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qA12_open_hash_set>

ADT skup realizovan zatvorenim hešovanjem, operacija *insert*.

13

**ZATVORENO HEŠOVANJE**: jedan od mehanizama razrešenja kolizija, *hash* tabela je niz gde je pozicija objekta o u nizu hash(o); svaka ćelija u *hash* tabeli je: EMPTY, OCCUPIED (zauzeta, sadrži element) ili DELETED (nekad sadržala element koji je sada obrisan); dodavanje novog elementa n: 1) računa se hash vrednost za n; 2) ako je table[hash] EMPTY tada table[hash]=n i prelazi u OCCUPIED; 3) ukoliko je *pozicija zauzeta* određuje se *nova pozicija po nekoj strategiji*: a) linearno probavanje (hash(o)+1, hash(o)+2...) – *uzrokuje klastere* (usporava pretraživanje), b) kvadratno probavanje (hash(o)+1, hash(o)+4, hash(o)+9...) – traži se *prva dostupna ćelija sa statusom EMPTY*, usled čega je lanac kolizija uslovljen *strategijom probavanja* i zato može sadržati elemente različitih vrednosti *hash* f-je.

**Operacije**: 1) **pretraživanje** tabele, odnosno lanca kolizija za dati element, 2) **dodavanje** novog elementa u tabelu – na kraj lanca kolizija ili u prvu DELETED ćeliju, 3) **brisanje** iz tabele – lenjo brisanje (status ćelije se menja u DELETED).

**Lanac kolizija**: *hash* tabela je niz dužine M (veliki prost broj) pa zato lanac kolizija ima maksimalnu dužinu – kod *kvadratnog probavanjа* se za maksimalnu vrednost uzima (M–1)/2; prvih (M–1)/2 kvadratnih probavanja su uvek *probavanja različitih lokacija u tabeli*, posle (M–1)/2 neuspešnih probavanja proširuje se tabela (takođe slučaj kada je opterećenje tabele veće od 70%, odnosno kada je više od 70% ćelija OCCUPIED).

**public class CHashSet<T> implements Set<T> { … }**: od polja postoji niz objekata, niz statusa (enum E/O/D) sa idx koji odgovaraju idx objekata i broj elemenata.

**OHashSet()**, **OHashSet(size)**, **hash(T o)**, **reset()**: konstruktori, hash f-ja i anuliranje polja.

**int searchCollisionChain(T element, int hashValue)**: hashValue postaje currentPosition, prati se brojač i uzima se maxChainLength = (table.length – 1) / 2; prolazak petljom kroz lanac i vraćanje currentPosition ukoliko se pronađe element, u suprotnom se vraća -1.

**boolean insert(T element)**: uzimaju se *hash* elementa i maksimalna dužina lanca (M–1)/2, prati se brojač i, kraj lanca i prva dostupna pozicija; kretanje kroz petlju dok se ne dođe do kraja lanca ili dok brojač ne dostigne maksimalnu dužinu – ako je pozicija zauzeta ekvivalentnim elementom vraća se false, inače se proverava da li je već pronađena prva dostupna pozicija (ako nije – uzima se trenutna) i da li se došlo do kraja lanca kolizije. Nakon petlje ukoliko ne postoji dostupna pozicija ili je opterećenje veće od 70% tabela se proširuje i element se dodaje metodom add(element, hashValue), a ukoliko postoji se dodaje.

**void add(T element, int hashValue)**: kretanje kroz lanac na sličan način kao sa insert(T element) metodom, ali ovaj put bez ijedne provere osim da li je pronađena dostupna pozicija.

**void expand()**: veličina tabele je najmanji prost broj koji je barem dva puta veći nego što je trenutna veličina tabele; postupak: 1) napravi se kopija stare tabele i statusa; 2) realociramo tabelu i postavimo statuse svih ćelija na EMPTY; 3) iteriramo kroz staru tabelu i ćelije u statusu OCCUPIED dodajemo u novu tabelu; 4) staru tabelu i statuse postavimo na null vrednosti.

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p05_hash_tables>

ADT skup realizovan zatvorenim hešovanjem, operacije *member* i *remove*.

14

**int searchCollisionChain(T element, int hashValue)**: uzima se *hash* elementa koji se traži kao trenutna pozicija, prate se brojač i maksimalna dužina lanca (M–1)/2; kretanje kroz while dok je brojač manji od maks. dužine i dok je status trenutne pozicije != EMPTY – ako je status OCCUPIED proverava se sadržaj elementa i ako su jednaki vraća se trenutna pozicija, u suprotnom se ažuriraju brojač i trenutna pozicija; na kraju se vraća trenutna pozicija ili -1 ukoliko element nije pronađen.

**boolean member(T element)**: prosto se iskoristi se pretraga lanca kolizije.

**boolean remove(T element)**: koristi se pretraga lanca kolizije za nalaženje pozicije traženog elementa; ukoliko je element pronađen status sa njegovom pozicijom se postavi na Status.DELETED a broj elemenata se smanji.

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p05_hash_tables>

ADT mapa realizovana otvorenim hešovanjem, operacije *insert* i *delete*.

15

**ADT MAPA, OTVORENO HEŠOVANJE**: čvorovi u tabeli sada sadrže parove (ključ, vrednost), a *hash* f-ja se primenjuje samo na ključeve (drugim rečima, lanac kolizija sadrži samo elemente čiji ključevi imaju istu vrednost *hash* f-je).

**private static class Node { … }**: *wrapper* klasa za realizaciju lanaca kolizije, sadrži Object key, Object value i Node next pokazivače.

**OHashMap()**, **OHashMap(size)**, **hash(T o)**: konstruktori i hash f-ja (hashCode() % table.length).

**Node[] searchCollisionChain(K key, int hashValue)**: lista kolizija se pretražuje tražeći čvor sa odgovarajućim ključem; ukoliko takav čvor postoji u tabeli vraća se niz od dve reference, i to na traženi čvor, kao i na onaj koji mu prethodi.

**boolean insert(K key, V value)**: pretražuje se lista kolizija za *hash* vrednost ključa; ukoliko se takav ključ pronađe vraća se false, inače se kreira novi element i postavlja na početak liste.

**boolean delete(K key)**: pretražuje se lista kolizija za *hash* vrednost ključa; ukoliko se takav ključ ne pronađe vraća se false, inače se pitamo da li je ključ na početku liste – ako jeste, prevezuje se sam početak liste, a ako nije, prevezujemo rezultat pretrage liste njemu prethodnim elementom.

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p05_hash_tables>

ADT mapa realizovana otvorenim hešovanjem, operacije *get* i *modify*.

16

**V get(K key)**: u praksi se često dešava da postoje elementi koji se dobavljaju češće u odnosu na prosek (*hot* i *cold data*) – primenjuje se LRU strategija (*least recently used* – poslednji traženi element pomeri se na početak lanca kolizija). Pretražuje se lista kolizija za *hash* vrednost ključa i ukoliko se ne pronađe vraća se null; inače, koristeći oba pokazivača koje vraća pretraga traženi element se prevezuje na početak liste, nakon čega se kao rezultat vraća početak liste kolizija.

**boolean modify(K key, V value)**: pretražuje se lista kolizija za *hash* vrednost ključa; ukoliko se takav ključ ne pronađe vraća se false, inače postavljamo vrednost na argument sa kojim je pozvan metod i vraćamo true.

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p05_hash_tables>

Pretraživanje sa vraćanjem – opšti opis postupka i modifikacije.

01

**PRETRAŽIVANJE SA VRAĆANJEM**: algoritamski postupak – ideja ili šablon primenjiv na širi spektar algoritamskih problema, *backtracking* je algoritamski postupak primenjiv na široku klasu problema poput 1) kombinatorne enumeracije – pronalaženje *svih konfiguracija* sistema koje *zadovoljavaju određeni kriterijum* (primeri: svi podskupovi skupa pozitivnih brojeva čiji je zbir jednak broju T, svi rasporedi 8 kraljica na šahovskoj tabli tako da se kraljice međusobno ne napadaju) i 2) kombinatorne optimizacije – pronalaženje konfiguracije sistema koja zadovoljava određeni *optimizacioni kriterijum* (primer: problem trgovačkog putnika – naći najkraću turu na mapi gradova).

**Konfiguracija** kombinatornog sistema može se predstaviti vektorom/nizom C = (c0, c1, ..., ct) čije elemente nazivamo komponentama (primer: kraljice na tabli – vektor od 8 komponenti koje su tada koordinate polja na kojima su kraljice). **Konfiguracija podskupa** skupa S – logički vektor C od k komponenti, gde je k kardinalnost skupa S; C[i] = true / false → S[i] jeste / nije u podskupu. **Parcijalna konfiguracija kombinatornog sistema** – vektor Ck, k < t, kod kojeg su prvih k komponenti poznate (određeni), dok su ostale nepoznate (neodređene).

***BACKTRACKING***: problem – pronaći sve konfiguracije C koje zadovoljavaju logički predikat P; osnovna ideja: 1) na osnovu P odredimo predikat Q koji je primenjiv na parcijalne konfiguracije sistema kojim definišemo da li je neka parcijalna konfiguracija Ck proširiva (primeri: kod zbira Q = postoji Ck+1 u kojoj je zbir elemenata ≤ T, kod kraljica Q = postoji Ck+1 u kojoj se kraljice ne napadaju), 2) parcijalne konfiguracije koje ne zadovoljavaju Q zovemo konfiguracijama neuspeha, 3) a one koje zadovoljavaju Q zovemo potencijalnim konfiguracijama; *backtracking* je dakle sistematično (iscrpno, sveobuhvatno) proširivanje potencijalnih konfiguracija.

**STABLO *BACKTRACK*-A**: proširivanje potencijalnih konfiguracija se može predstaviti *backtracking* stablom, gde je koren stabla prazna konfiguracija, unutrašnji čvorovi stabla potencijalne konfiguracije, listovi ciljne (koje zadovoljavaju P) ili konfiguracije neuspeha; ako je čvor A sin čvora B – konfiguracija A se dobija proširivanjem potencijalne konfiguracije B.

***BACKTRACKING*, DALJE**: *backtracking –* „obilazak“ stabla u dubinu: 1) ne konstruiše se stablo *backtrack*-a već postoji jedna konfiguracija koja se menja **proširivanjem** i **sažimanjem**; 2) dinamika promene te parcijalne konfiguracije korespondira obilasku stabla *backtrack*-a u dubinu; 3) parcijalna konfiguracija se proširuje u ciljnu (konfiguracija koja zadovoljava P) – imamo rešenje; 4) ako se parcijalna ne može proširiti tada se **sažima i nastavlja se sa proširenjima sažete parcijalne konfiguracije** (sažimanje – izbacivanje poslednje komponente iz parcijalne konfiguracije); 5) proširenje parcijalne konfiguracije ne uspeva iz dva razloga – ili su već ispitana sva proširenja, ili proširenje vodi u konfiguraciju neuspeha; 6) **sažimanje prve parcijalne konfiguracije** – **kraj *backtracking*-a**.

**Proširivanje**: mora se uvesti red prilikom proširivanja trenutne parcijalne konfiguracije kako se ne bi proširivala na neku konfiguraciju koja je već bila ispitana; za trenutnu parcijalnu konfiguraciju dužine k-1 formira se Dk: lista dopustivih elemenata za k-tu komponentu (kako imamo t komponenti to će biti t listi dopustivih elemenata); trenutna parcijalna konfiguracija Ck-1 se proširuje tako što: 1) f = prvi element liste Dk, 2) obriše se f iz Dk, 3) formira se Ck proširenjem Ck-1 sa f. *Backtrack* nerekurzivno:

Vector C; // konfiguracija sistema

void backtracking() {

D[] = new D[C.length]; // liste dopustivih elemenata

int k = 0; // trenutna komp. (dužina parc. konfig.)

compute D[0]; // odredi dopustive el. za prvu komp.

while (k >= 0) {

while (!D[k].empty) {

f = remove first element in D[k];

C[k] = f;

if (k == C.length – 1)

printConfiguration(); // imamo rešenje

else {

k++; // idemo na sledeću komponentu

compute D[k]; } } // odredi dopustive el. za sledeću komp.

k--; } } // vraćamo se nazad

// Određivanje dopustivih el. na osnovu predikata Q primenjenog na trenutnu parcijalnu konfiguraciju.

**Algoritam *backtracking*-a**: počinje se od vektora konfiguracije sistema C; računa se lista dopustivih elemenata za k = 0, gde je k dužina parcijalne konfiguracije, odnosno njena trenutna komponenta; ulazi se u while petlju koja traje sve dok se parcijalna konfiguracija ne sažme nazad do početne, odnosno sve dok je k ≥ 0; unutar petlje se uzima prvi dopustiv element i njime se fiksira k-ta komponenta parcijalne konfiguracije; sledi provera da li je dužina parcijalne konfiguracije jednaka dužini vektora C – ako jeste znači da imamo rešenje, a ako nije prelazi se na sledeću komponentu i računa se lista dopustivih za nju; na kraju se, unutar while petlje, k smanjuje za jedan, odnosno radi se sažimanje ukoliko nije preostalo dopustivih elemenata za trenutnu parcijalnu konfiguraciju. *Backtracking* rekurzivno:

Vector C; // trenutna konfiguracija sistema

void backtracking() {

backtracking(0); }

backtracking (int k) { // k – trenutna komponenta

compute D; // određivanje liste dopustivih elemenata za k-tu komp.

while (!D.empty()) {

f = remove first element in D;

C[k] = f;

if (k == C.length – 1)

printConfiguration();

else

backtracking (k+1); } }

Računanje dopustivih elemenata je moguće optimizovati: ukoliko za komponente C znamo da su u nekom intervalu [min, max] tada umesto liste dopustivih elemenata čuvamo samo jedan, i to najmanji dopustivi element, dok se sledeći određuje na osnovu prethodnog (primer: kraljice – vrste/kolone su u intervalu [0 .. 7]:

Vector C;

backtracking () {

Vector D;

int k = 0;

D[0] = first available element >= min

while (k >= 0) {

while (D[k] <= max) { // dok D[k] nije uzeo max dopustivu vrednost…

C[k] = D[k] // C[k] se fiksira sa D[k]

D[k] = first available element > D[k] or max + 1 (no available elements)

if (k == C.length – 1) printConfiguration();

else {

k++;

D[k] = first available element >= min or max + 1 (no available elements) } }

k--; } }

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p06_backtrack>

Rešenje problema kraljica pretraživanjem sa vraćanjem (objasniti i implementirati bar jednu verziju algoritma po izboru).

02

**PROBLEM N KRALJICA**: rasporediti N kraljica na šahovsku tablu dimenzija N \* N tako da nema kraljica koje se međusobno napadaju (a napadaju se ako: 1) x1 = x2, 2) y1 = y2, 3) |x1 – x2| = |y1 – y2|) – koliko ima takvih rasporeda i koji su (rešenje po vrstama)? Rešenje se predstavlja nizom int[] table, gde su kraljice na pozicijama (i, table[i]), i < n.

**VERZIJA 1 – LISTE DOPUSTIVIH POLJA**: unutrašnja klasa AvailableCell *wrapper* za položaj u vrsti sa pokazivačem na sledeći (lista dopustivih polja), polja klase su veličina table n, brojač rešenja slCounter, raspored kraljica int[] table i niz listi dopustivih polja; konstruktor prima veličinu n i inicira table i availableCells, printSolution() uveličava broj rešenja i štampa trenutno; find() metod: 1) formiranje liste raspoloživih polja za prvu vrstu[[3]](#footnote-3) i postavljanje currentRow na 0; 2) pokreću se dve while petlje, gde se prva prekida kada se konfiguracija sažme na manje od nule (odnosno kada smo isprobali sve konfiguracije), a druga kada više nema raspoloživih polja; 3) table[currentRow] uzima vrednost prvog raspoloživog polja, koren raspoloživih polja prelazi na sledeći element[[4]](#footnote-4); 4) proverava da li postoji rešenje sa (currentRow == n – 1) – ako da štampa se i beleži, ako ne izvrši se currentRow++ i traži nova lista raspoloživih polja; findAvailableCells(int currentRow) metod: 1) postavlja listu za trenutni currentRow na null; 2) kretanje kroz dve for petlje, gde prva ide od poslednje ka prvoj koloni (jer su tako popunjene dosadašnje kraljice), a druga od prve kolone ka poslednjoj i proverava da li su polja u koliziji sa već postavljenim kraljicama po uslovu #1 (table[i] == j – provera po vertikali) i uslovu #3 (|i – currentRow| == |table[i] – j| - provera po dijagonali, horizontala pa vertikala) – uslov #2 se ne proverava jer su vertikale već „ograđene“ postavkom for petlji; 3) ako je polje prošlo proveru dodaje se na početak liste (polja su tako sortirana neopadajuće).

**VERZIJA 2 – ČUVANJE JEDNOG DOPUSTIVOG ELEMENTA**: umesto availableCells sada postoji int[] firstAvailableCell (sadrži prvo dostupno polje za vrstu i); find() metod: 1) formiranje liste raspoloživih polja za prvu vrstu prosto se postavi firstAvailableCell[0] na 0; 2) druga while petlja se sada prekida kada je firstAvailableCell[currentRow] < n; 3) table[currentRow] uzima vrednost prvog raspoloživog polja, currentRow se povećava i pokreće se updateFirstAvailableCell(currentRow); 4) u else grani takođe postoji ažuriranje dopustivog polja; updateFirstAvailableCell(int currentRow) metod: 1) kretanje kroz dve for petlje, gde prva ide od prvog dostupnog polja za trenutni red do kraja reda, a druga od prvog reda do trenutnog; 2) druga petlja proverava koliziju sa već postavljenim kraljicama; 3) ukoliko provera prođe, uzima se trenutna vrednost brojača prve for petlje za prvo raspoloživo polje; 4) ukoliko nijedna ćelija nije dostupna, vrednost prvog raspoloživog polja se stavlja na n (izlaz iz traženja rešenja).

**VERZIJA 3 – REKURZIVNO**: prvo dostupno polje sada se računa tokom traženja rešenja; find() metod: 1) izlaz je currentRow == n; 2) prolazak kroz čitav red od 0 do n brojačem j, provera da li je polje (currentRow, j) dostupno i ako jeste uzima se i traži se rešenje za currentRow + 1; availableCell(int currentRow, int currentColumn) metod: 1) prolazak for petljom od 0 do currentRow; 2) provera kolizije za vrednost table[i].

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p06_backtrack>

Rešenje problema ruksaka pretraživanjem sa vraćanjem (objasniti i implementirati bar jednu verziju algoritma po izboru).

03

**PROBLEMI KOMBINATORNE OPTIMIZACIJE**: probleme kombinatorne optimizacije često je moguće formulisati na opšti način: 1) dat je skup S od n objekata, 2) svaki objekat ima neku vrednost (težinu, cenu, profit) predstavljenu f-jom f: S → Integer, 3) f-ja f: S → Integer je definisana i na partitivnom skupu skupa S i aditivna je: f(T) = zbir f(x) za svako x iz T, gde je T ⊂ S, 4) potrebno je naći podskup skupa S koji zadovoljava određena ograničenja definisana logičkim predikatom P tako da se f maksimizuje (ili minimizuje).

**PROBLEM RUKSAKA**: dat je skup S od n predmeta pri čemu svaki ima težinu (*weight*) i cenu (*profit*); dat je ranac u koji se smeštaju predmeti čija težina ne prevazilazi maksimalnu težinu knapsackWeight; problem: napuniti ranac predmetima iz S tako da se maksimizuje profit, što se u opštem slučaju ne dešava gramzivim punjenjem. Opšta formulacija:

Set greedy(Set S) {

Sort S in non-increasing order by f or some function derived from f

Set solution = empty set

for (int i = 0; i < S.length; i++) {

if (solution U {S[i]} satisfies P) {

solution = solution U {S[i]} } } }

return solution }

Ideja gramzivih algoritama je da lokalno optimalne odluke vode globalno optimalnom rešenju, kao i da se jednom doneta odluka više ne može opozvati; gramzive strategije za punjenje ranca: 1) sortiranje po profitu, ili 2) po težini (najmanja ka najvećoj), ili 3) po „gustini profita“ (*profit / weight*); konkretan primer dat na slajdovima sa predavanja.

**BACKTRACK REŠENJE**: tražimo sve podskupove T skupa predmeta S koji zadovoljavaju kriterijum maksimalne težine ruksaka, pri čemu trenutni podskup T čuvamo ukoliko je profit ostvaren sa T veći od trenutno maksimalnog profita; konfiguracija se predstavlja logičkim nizom: in[k] = true / false → k-ti predmet jeste/nije u ruksaku; dopustivi elementi za k-tu komponentu su: true (ako se k-ti predmet može smestiti u ranac – zbir težina prethodno ubačenih predmeta sa težinom k-tog je manja ili jednaka od maksimalno dozvoljene), ili false (uvek).

Rešenje problema ruksaka algoritmom grana i granica.

04

**ALGORITAM GRANA I GRANICA (*BRANCH AND BOUND*)**: radi se o modifikaciji *backtracking* postupka za probleme kombinatorne optimizacije; osnovna ideja: 1) kreće se od trenutne konfiguracije sistema Ck-1, gde je Dk skup dopustivih elemenata za k-tu komponentu; 2) za svako d iz Dk ocenjuje se gornje ograničenje (*bound*) f-je f u ciljnim konfiguracijama koje se dobijaju iz Ck-1 proširenim sa d; 3) proširenje Ck-1 sa d se isplati samo ako je bound(d) ≤ trenutnog maksimalnog f.

***BRANCH AND BOUND* REŠENJE RUKSAKA**: postoji neko prethodno ustanovljeno trenutno maksimalno rešenje M; za k-ti predmet i trenutnu parcijalnu konfiguraciju potrebno je utvrditi: **1)** koliko je gornje ograničenje maksimalnog profita PA ubacivanjem predmeta, pod uslovom da se može dodati; **2)** koliko je gornje ograničenje PS neubacivanjem predmeta; proverava se PA > M i ako je tačno, predmet se ubaci i nastavlja se *backtrack*, a ako nije ne ubacuje se i *backtrack* ide dalje; ključno: gornje ograničenje se može ustanoviti gramzivim algoritmom.

**VEZA GRAMZIVOG SA *BRANCH AND BOUND* ALGORITMOM**: problem ruksaka se u slučaju deljivih predmeta lako rešava gramzivim algoritmom; svaka konfiguracija koja se dobija kod 1/0 rešenja ruksaka može se dobiti i rešavanjem gramzivim algoritmom, zato je profit ostvaren selekcijom nedeljivih predmeta uvek manji ili jednak profitu selekcijom deljivih.

URL

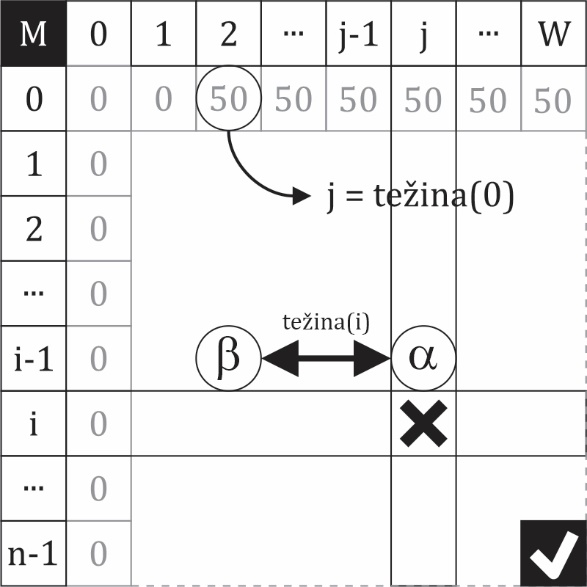
Rešenje problema ruksaka dinamičkim programiranjem.

07

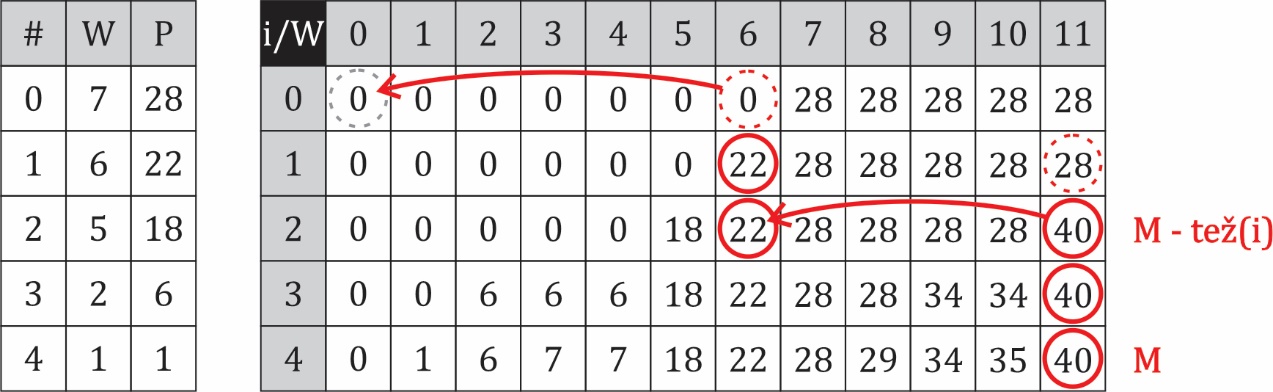
**DINAMIČKO PROGRAMIRANJE**: algoritamski postupak primenjiv na široku klasu problema optimizacije, podrazumeva popunjavanje tabele u procesu rešavanja optimizacionog problema; na ovaj način se rešavaju problemi koji zadovoljavaju *Bellman*-ov princip optimalnosti: problem P ima optimalnu strukturu ako se optimalno rešenje P može dobiti iz optimalnih rešenja manjih instanci problema P (kod ruksaka je ta dimenzionalnost opisana brojem predmeta i kapacitetom ranca); dinamičko programiranje je popunjavanje tabele optimalnih rešenja **svih** instanci problema P(a1, a2, ... , an), nakon čega optimalno rešenje dobijamo iz optimalnih rešenja manjih instanci P zbog čega mora postojati relacija između optimalnog rešenja celog problema i njegovih manjih instanci, a takođe i trivijalne instance od P (kod ruksaka su to situacije sa samo jednim predmetom za proizvoljni kapacitet, i proizvoljan broj predmeta za ruksak kapaciteta 0).

**POSTUPAK**: **1)** kreira se tabela maks. profita sa [0 … n-1] vrsta (predstavljaju predmete) i [0 … W] kolona (svi kapaciteti od 0 do W); **2)** posmatrajmo proizvoljni element matrice M[i][j] – ovaj element nam govori koliki profit se može ostvariti ubacivanjem od 0-tog do i-tog predmeta (uključujući i-ti) u ruksak kapaciteta j; **3)** rešenje problema je poslednji element M[n-1][W] koji se dobija ubacivanjem svih predmeta u ruksak kapaciteta W – ideja je da se taj element dobije na osnovu prethodno popunjenih; **4)** rekurzivni odnos se uspostavlja na sledeći način:

Prva opcija je da i-ti predmet ne može da stane u ruksak kapaciteta j tada se optimalno punjenje dobija posmatranjem svih predmeta osim i-tog za ruksak istog kapaciteta; druga opcija je da i-ti predmet može da stane u ruksak i tada se uzima maksimum od profita u slučaju kada se predmet namerno ne uzme i onog u slučaju kada se uzme; nakon postavljanja rekurzivnog odnosa potrebno je odrediti trivijalne slučajeve: **1)** M[i][0] = 0, ∀i ∈ [0, n - 1], **2)** uzmimo npr. da za 0-ti predmet važi težina(0) = 2, profit = 50 – tada se prva vrsta popunjava na način dat u tabeli**:**

****

**REKONSTRUKCIJA STRUKTURE REŠENJA**: rešenje se rekonstruiše posmatranjem tabele maksimalnog profita: ako je MP[i][j] = MP[i -1][j] tada i-ti predmet nije ubačen u ruksak kapaciteta j, a ako nije od trenutne vrednosti profita se oduzima težina trenutnog predmeta i prelazi se na prethodni:



URL

Računanje string *edit* distance dinamičkim programiranjem.

08

**STRING DISTANCE**: reč je o funkciji D(s1, s2), D: String × String → ℝ za koju važi: **1)** D(s1, s2) = 0 → s1 i s2 su identični; **2)** D(s1, s2) = D(s2, s1); **3)** D(s1, s2) < D(s1, s2) → s1 je sličniji stringu s2 nego s3. String distance možemo klasifikovati u: **1)** edit distance – računamo stepen transformacije jednog stringa u drugi, pri čemu veći stepen transformacije znači i veću distancu (odnosno manju sličnost); **2)** set distance – računamo stepen preklapanja stringova, pri čemu veći stepen preklapanja znači i manju distancu (veću sličnost).

**N-GRAM DISTANCE**: klasa jednostavnih set string distanci; postupak: **1)** od stringa se formira skup n-grama (primer: „algoritam“ → „alg“, „lgo“, … , „ita“, „tam“), **2)** sličnost se meri kao stepen preklapanja n-gram skupova stringova za neko fiksirano n; računanje *Jaccard*-ove n-gram distance:

**EDIT STRING DISTANCA**: najjednostavnija edit string distanca meri broj elementarnih transformacija potrebnih da se jedan string transformiše u drugi, pri čemu su elementarne operacije dodavanje, brisanje i supstitucija karaktera; računanje edit distance je optimizacioni problem – traži se minimalni broj elemenata transfromacija (primer: „petar“ → „patrik“, 4 transformacije).

1. Drugim rečima, *Comb* prolaz se za k = 1 ponavlja dok postoje elementi u inverziji što je zapravo *Bubble sort*. [↑](#footnote-ref-1)
2. Kolizije i “*birthday paradox*”: 367 ljudi u sobi za 100% verovatnoće da su dve rođene istog dana, 70 za 99,99%, 35 za 80%, 23 za 50%. [↑](#footnote-ref-2)
3. Praktično slaže sva polja prve vrste u niz od poslednjeg ka prvom. [↑](#footnote-ref-3)
4. Ovo je važno ukoliko dođe do povratka na ovo polje jer će se tako isprobati drugačije polje nego prvi put [↑](#footnote-ref-4)