Programiranje 2

Pretraživanje i sortiranje

Milena Vujošević Janičić
Jelena Graovac

www.matf.bg.ac.rs/~milena
www.matf.bg.ac.rs/~jgraovac

Programiranje 2
Beograd, 12. mart, 2020.

Pregled

Pregled

Poređenje i poredak

- U mnogim problemima potrebno je uporediti dva objekta. U nekim situacijama potrebno je proveriti da li su dva objekta jednaka, a u nekim da li je jedan manji (ili veći) od drugog.
- Na primer, jednakost niski moguće je proveriti sledećom funkcijom:

```
int jednake_niske(const char a[], const char b[]) {
  int i = 0;
  while (a[i] == b[i]) {
    if (a[i] == '\0') return 1;
    i++;
  }
return 0;
}
```

- Jednakost niski moguće je proveriti i funkcijom strcmp iz standardne biblioteke, kojom se vrši provera leksikografskog poretka dve niske.

Poređenje i poredak

- Za dve vrednosti tipa neke strukture, provera jednakosti svodi se na proveru jednakosti svih članova pojedinačno ili možda na neki drugi način. Na primer, dva razlomka nisu jednaka samo ako su im i imenilac i brojilac jednaki, nego i u nekim drugim slučajevima.

```
typedef struct razlomak {
  int brojilac;
  int imenilac;
} razlomak;

int jednaki_razlomci(const razlomak *a, const razlomak *b) {
  return (a->imenilac * b->brojilac == b->imenilac * a->brojilac);
}
```

Poređenje i poredak

- Na primer, naredna funkcija poredi dvoslovne oznake država po standardu ISO 3166 3 . Ona vraća vrednost -1 ako je prvi kôd manji, 0 ako su zadati kôdovi jednaki i 1 ako je drugi kôd manji:

```
int poredi_kodove_drzava(const char *a, const char *b){
  if (a[0] < b[0])
    return -1;
  if (a[0] > b[0])
    return 1;
  if (a[1] < b[1])
    return -1;
  if (a[1] > b[1])
    return 1;
  return 0;
}
```

Poređenje i poredak

- Slično se mogu porediti i datumi opisani narednom strukturom:

```
typedef struct datum {
  unsigned dan;
  unsigned mesec;
  unsigned godina;
} datum;

int poredi_datume(const datum *d1, const datum *d2){
  if (d1->godina < d2->godina)
    return -1;
  if (d1->godina > d2->godina)
    return 1;
  if (d1->mesec < d2->mesec)
    return -1;
  if (d1->mesec > d2->mesec)
    return 1;
  if (d1->dan < d2->dan)
    return -1;
  if (d1->dan > d2->dan)
    return 1;
  return 0;
}
```

Poređenje i poredak

- Može se napisati i jedinstven logički izraz kojim se proverava da li je prvi datum ispred drugog:

```
int datum_pre(const datum *d1, const datum *d2){
  return d1->godina < d2->godina ||
         (d1->godina == d2->godina && d1->mesec < d2->mesec) ||
         (d1->godina == d2->godina && d1->mesec == d2->mesec &&
         d1->dan < d2->dan);
}
```

Poređenje i poredak

- Niske, osim leksikografski, mogu da se porede i na neki drugi način, na primer, samo pod dužini:

```
int poredi_niske(const char *a, const char *b){
  return (strlen(a)-strlen(b));
}
```

- Dva razlomka (čiji su imenioci pozitivni) mogu da se porede sledećom funkcijom:

```
int poredi_razlomke(const razlomak *a, const razlomak *b){
  return (a->brojilac*b->imenilac - b->brojilac*a->imenilac);
}
```

- Ako je zadata relacija poretka (ili strogog poretka), onda se može proveriti da li je niz uređen (ili sortiran) u skladu sa tom relacijom.

Pregled

Pretraživanje

- Pod pretraživanjem za dati niz elemenata podrazumevamo određivanje indeksa elementa niza koji je jednak datoj vrednosti ili ispunjava neko drugo zadato svojstvo (npr. najveći je element niza).

Linearno pretraživanje

- *Linearno* (ili *sekvencijalno*) pretraživanje niza je pretraživanje zasnovano na ispitivanju redom svih elemenata niza ili ispitivanju redom elemenata niza sve dok se ne naiđe na traženi element (zadatu vrednost ili element koji ima neko specifično svojstvo)
- Linearno pretraživanje je vremenske linearne složenosti po dužini niza koji se pretražuje.
- Ukoliko se u nizu od n elemenata traži element koji je jednak zadatoj vrednosti, u prosečnom slučaju (ako su elementi niza slučajno raspoređeni), ispituje se $n/2$, u najboljem 1, a u najgorem n elemenata niza.

Primer

- Naredna funkcija vraća indeks prvog pojavljivanja zadatog celog broja x u zadatom nizu a dužine n ili vrednost -1, ako se taj broj ne pojavljuje u a:

```
int linearna_pretraga(int a[], int n, int x){
  int i;
  for (i = 0; i < n; i++)
    if (a[i] == x)
      return i;
  return -1;
}
```

- Složenost ove funkcije je O(n)

Primer

Česta greška prilikom implementacije linearne pretrage je prerano vraćanje negativne vrednosti:

```
int linearna_pretraga(int a[], int n, int x) {
  int i;
  for (i = 0; i < n; i++)
    if (a[i] == x)
      return i;
    else
      return -1;
}
```

Navedena naredba `return` prouzrokuje prekid rada funkcije tako da implementacija može da prekine petlju već nakon prve iteracije.

Primer

Linearna pretraga može biti realizovana i rekurzivno. Pozivom `linearna_pretraga(a, i, n, x)` nalazi se indeks prvog pojavljivanja elementa x u nizu `a[i, n-1]`. Kako bi se izvršila pretraga celog niza, potrebno je izvršiti poziv `linearna_pretraga(a, 0, n, x)`, pri čemu je n dužina niza a:

```
int linearna_pretraga(int a[], int i, int n, int x){
  if (i == n)
    return -1;
  if (a[i] == x)
    return i;
  return linearna_pretraga(a, i+1, n, x);
}
```

Primer

Ukoliko se zadovoljimo pronalaženjem poslednjeg pojavljivanja elementa x, kôd se može dodatno uprostiti:

```
int linearna_pretraga(int a[], int n, int x){
  if (n == 0)
    return -1;
  else if (a[n - 1] == x)
    return n-1;
  else return linearna_pretraga(a, n-1, x);
}
```

Primer

U oba slučaja, rekurzivni pozivi su repno rekurzivni. Eliminacijom repne rekurzije u drugom slučaju dobija se:

```
int linearna_pretraga(int a[], int n, int x){
  while (n > 0) {
    if (a[n - 1] == x)
      return n - 1;
    n--;
  }
  return -1;
}
```

Primer

Naredna funkcija vraća indeks najvećeg elementa medu prvih n elemenata niza a (pri čemu je n veće ili jednako 1):

```
int max(int a[], int n){
  int i, index_max;
  index_max = 0;
  for(i = 1; i < n; i++)
    if (a[i] > a[index_max])
      index_max = i;
  return index_max;
}
```

Primer

Linearno pretraživanje može se koristiti i u situacijama kada se ne traži samo jedan element niza sa nekim svojstvom, nego više njih. Sledeći program sadrži funkciju koja vraća indekse dva najmanja (ne nužno različita) elementa niza. Ona samo jednom prolazi kroz zadati niz i njena složenost je O(n).

Primer

```
#include <stdio.h>

int min2(int a[], int n, int *index_min1, int *index_min2)
{
    int i;
    if (n < 2)
      return -1;
    if (a[0] <= a[1]) {
      *index_min1 = 0; *index_min2 = 1;
    }
    else {
      *index_min1 = 1; *index_min2 = 0;
    }
    for (i = 2; i < n; i++) {
      if (a[i] < a[*index_min1]) {
        *index_min2 = *index_min1;
        *index_min1 = i;
      } else if (a[i] < a[*index_min2])
        *index_min2 = i;
    }
    return 0;
}

int main()
{
    int a[] = {12, 13, 2, 10, 34, 1};
    int n = sizeof(a)/sizeof(a[0]);
    int i, j;
    if (min2(a, n, &i, &j) == 0)
      printf("Najmanja dva elementa niza su %d i %d.\n", a[i], a[j]);
    else
      printf("Neispravan ulaz.\n");
    return 0;
}
```

Binarna pretraga

- *Binarno* pretraživanje je pronalaženje zadate vrednosti u zadatom skupu objekata, pri čemu se pretpostavlja da su objekti zadatog skupa sortirani, i u svakom koraku, sve dok se ne pronađe tražena vrednost, taj skup se deli na dva dela i pretraga se nastavlja samo u jednom delu — odbacuje se deo koji sigurno ne sadrži traženu vrednost.
- Binarno pretraživanje ne možemo da primenimo ukoliko skup objekata koji pretražujemo nije sortiran.
- Primer: pogađanje brojeva sa poznatom gornjom granicom
- Primer: pogađanje brojeva ukoliko gornja granica nije poznata
- Složenost $O(\log n)$
- Razlog zašto su rečnici, enciklopedije, telefonski imenici sortirani (varijanta pretraživanja koja se naziva *interpolaciona pretraga*)

Binarna pretraga

Binarno pretraživanje može se implementirati iterativno ili rekurzivno. Naredna funkcija daje rekurzivnu implementaciju binarnog pretraživanja. Poziv `binarna_pretraga(a, l, d, x)` vraća indeks elementa niza a izmedu l i d (uključujući i njih) koji je jednak zadatoj vrednosti x ako takav postoji a -1 inače. Dakle, ukoliko se želi pretraga čitavog niza dužine n, funkciju treba pozvati sa `binarna_pretraga(a, 0, n-1, x)`.

```
int binarna_pretraga_(int a[], int l, int d, int x){
  int s;
  if (l > d)
    return -1;
  s = l + (d - l)/2;
  if (x == a[s])
    return s;
  if (x < a[s])
    return binarna_pretraga_(a, l, s-1, x);
  else /* if (x > a[s]) */
    return binarna_pretraga_(a, s+1, d, x);
}
int binarna_pretraga(int a[], int n, int x){
  return binarna_pretraga_(a, 0, n-1, x);
}
```

Binarna pretraga

Oba rekurzivna poziva u prethodnoj funkciji su repno-rekurzivna, tako da se mogu jednostavno eliminisati. Time se dobija iterativna funkcija koja vraća indeks elementa niza a koji je jednak zadatoj vrednosti x ako takva postoji i -1 inače.

```
int binarna_pretraga(int a[], int n, int x){
  int l, d, s;
  l = 0; d = n-1;
  while(l <= d) {
    s = l + (d - l)/2;
    if (x == a[s])
      return s;
    if (x < a[s])
      d = s - 1;
    else /* if (x > a[s]) */
      l = s + 1;
  }
  return -1;
}
```

Binarna pretraga

Da bi se smanjio broj poređenja, proveru na jednakost treba ostaviti za kraj:

```
int binarna_pretraga(int a[], int n, int x) {
   int l, d, s;
   l = 0; d = n-1;
   while(l <= d)  {
     s = l + (d - l)/2;
     if (x > a[s])
         l = s + 1;
     else if (x < a[s])
         d = s - 1;
     else /* if (x == a[s]) */
         return s;
   }
   return -1;
}
```

Binarno traženje prelomne tačke

Napisati funkciju koja vraća indeks prvog broja u sortiranom nizu a dimenzije n, koji je veći ili jednak broju x. Ukoliko su svi elementi niza manji od x, funkcija treba da vrati n.

```
int prvi_veci_ili_jednak(int a[], int n, int x){
  int l = 0, d = n;
  while (l < d) {
    int s = l + (d - l) / 2;
    if (a[s] < x) {
      l = s + 1;
    } else
      d = s;
    }
  return d;
}
```

Binarno traženje prelomne tačke

Primenom ove funkcije mogu se napisati i sledeće funkcije:

```
int binarna_pretraga(int a[], int n, int x){
  int p = prvi_veci_ili_jednak(a, n, x);
  if (p < n && a[p] == x)
    return p;
  return -1;
}

int broj_vecih_ili_jednakih(int a[], int n, int x){
  int p = prvi_veci_ili_jednak(a, n, x);
  return n - p;
}
```

Tehnika dva pokazivača

Razmotrimo problem određivanja broja parova različitih elemenata strogo rastuće sortiranog niza čiji je zbir jednak datom broju s. Naivna varijanta je da proverimo sve parove elemenata, što dovodi do algoritma složenosti $O(n^2)$. Bolja varijanta je da se za svaki element niza a i binarnom pretragom proveri da li se na pozicijama od i+1 do kraja niza nalazi element s-a i i ako postoji, da se uveća brojač parova. Time se dobija algoritam složenosti O(n log n). Međutim, postoji bolji algoritam i od toga.

Tehnika dva pokazivača

```
int broj_parova = 0;
int l = 0, d = n - 1;
while (l < d)
  if (a[l] + a[d] > s)
    d--;
  else if (a[l] + a[d] < s)
    l++;
  else {
    broj_parova++;
    l++; d--;
  }
```

Tehnika dva pokazivača

Ako je dat niz pozitivnih celih brojeva, odrediti koliko postoji nepraznih segmenata tog niza (podnizova uzastopnih elemenata) čiji elementi imaju zbir jednak datom broju s.

Naivno rešenje je ponovo da se ispitaju svi segmenti. Ako se za svaki segment iznova izračunava zbir, dobija se algoritam složenosti čak $O(n^3)$, koji je praktično neupotrebljiv. Ako se segmenti obilaze tako da se za fiksirani levi kraj l, desni kraj uvećava od l do n-1, onda se vrednost sume segmenata može izračunavati inkrementalno, dodajući a[d] na vrednost zbira prethodnog segmenta, što daje algoritam složenosti $O(n^2)$, koji je i dalje veoma neefikasan.

Važna tehnika primenljiva u mnogim zadacima je da se suma segmenta $\sum_{i=l}^{d} a_i$ izrazi kao $\sum_{i=0}^{d} a_i - \sum_{i=0}^{l-1} a_i$[6]. Pretpostavimo da umesto niza a znamo njegove parcijalne sume tj. da znamo niz p_k definisan sa $p_0 = 0$ i $p_k = \sum_{i=0}^{k-1} a_i$. Ako elemente niza p računamo inkrementalno, tokom učitavanja elemenata niza a, složenost tog izračunavanja je samo $O(n)$. Tada je suma segmenta na pozicijama $[l, d]$ jednaka $p_{d+1} - p_l$. Pošto su po uslovu zadatka brojevi u nizu a pozitivni, niz p je sortiran rastuće i ovim je zadatak sveden na zadatak da se u sortiranom nizu p pronađe broj elemenata čija je razlika jednaka zadatom broju, što je problem pomenut u prethodnom primeru (može se rešiti binarnom pretragom u ukupnoj složenosti $O(n \log n)$ ili tehnikom dva pokazivača u ukupnoj složenosti $O(n)$).

Tehnika dva pokazivača

Niz p nije neophodno čuvati u memoriji i tehnika dva pokazivača može da se primeni i na elemente originalnog niza.

```
/* broj segmenata trazenog zbira */
int broj = 0;
/* granice segmenta */
int l = 0, d = 0;
/* zbir segmenta */
int zbir = a[0];
while (1) {
  /* na ovom mestu vazi da je zbir = sum(a[l], ..., a[d]) i da
     za svako l <= d' < d vazi da je sum(a[l], ..., a[d']) < s */

  if (zbir < s) {
    /* prelazimo na interval [l, d+1] */
    d++;
    /* ako takav interval ne postoji, zavrsili smo pretragu */
    if (d >= n)
       break;
    /* na osnovu zbira intervala [l, d]
       izracunavamo zbir intervala [l, d+1] */
    zbir += a[d];
  } else {
    /* ako je zbir jednak trazenom,
       vazi da je sum(a[l], ..., a[d]) = s
       pa prijavljujemo interval */
    if (zbir == s)
       broj++;
    /* na osnovu zbira intervala [l, d]
       izracunavamo zbir intervala [l+1, d] */
    zbir -= a[l];
    l++;
```

Određivanje nula funkcije

Naredna funkcija pronalazi, za zadatu tačnost, nulu zadate funkcije f, na intervalu [l, d], pret- postavljajući da su vrednosti funkcije f u l i d različitog znaka.

```
float polovljenje(float (*f)(float), float l, float d, float epsilon){
  float fl=(*f)(l), fd=(*f)(d);
  for (;;) {
    float s = (l+d)/2;
    float fs = (*f)(s);
    if (fs == 0.0 || d-l < epsilon)
      return s;
    if (fl*fs <= 0.0) {
      d=s; fd=fs;
    }
    else {
      l=s; fl=fs;
    }
  }
}
```

Pregled

Sortiranje

- Sortiranje je jedan od fundamentalnih zadataka u računarstvu.
- Sortiranje podrazumeva uređivanje niza u odnosu na neko linearno uređenje.
- Primeri:
  - uređenje niza brojeva po veličini — rastuće ili opadajuće,
  - uređivanje niza niski leksikografski ili po dužini,
  - uređivanje niza struktura na osnovu vrednosti nekog polja.

Sortiranje

- Postoji više različitih algoritama za sortiranje nizova.
- Neki algoritmi su jednostavni i intuitivni, dok su neki kompleksniji, ali izuzetno efikasni.
- Najčešće korišćeni algoritmi za sortiranje su:
  - Bubble sort
  - Selection sort
  - Insertion sort
  - Shell sort
  - Merge sort
  - Quick sort
  - Heap sort
- `http://www.sorting-algorithms.com/`

Sortiranje

- Algoritmi sortiranja imaju različite vremenske i prostorne složenosti.
- Neki od algoritama za sortiranje rade u mestu (engl. in-place), tj. sortiraju zadate elemente bez korišćenja dodatnog niza.
- Drugi algoritmi zahtevaju korišćenje pomoćnog niza ili nekih drugih struktura podataka.
- Prilikom izračunavanja vremenske složenosti algoritama sortiranja obično se uzimaju u obzir samo operacije poređenja i operacije zamene, jer su one za podatke netrivijalnih tipova najskuplje operacije.

Sortiranje — klase složenosti

- Jednostavniji, sporiji algoritmi sortiranja imaju složenost najgoreg slučaja $O(n^2)$ i u tu grupu spadaju *bubble*, *insertion* i *selection*.
- *Shell sort* je algoritam kojem, u zavisnosti od implementacije, složenost varira od $O(n^2)$ do $O(n \log^2 n)$.
- Kompleksniji, brži algoritmi imaju složenost najgoreg slučaja $O(n \log n)$ i u tu grupu spadaju *heap* i *merge sort* (ovi algoritmi zahtevaju dodatni memorijski prostor).
- Algoritam *quick sort* ima složenost najgoreg slučaja $O(n^2)$, ali, pošto je složenost prosečnog slučaja $O(n \log n)$ i pošto u praksi pokazuje dobre rezultate, ovaj se algoritam ubraja u grupu veoma brzih algoritama.

Selection sort

- Primer animacije
- Ideja: pronaći najmanji element u nizu i postaviti ga na nulto mesto, zatim pronaći najmanji od ostatka niza i dovesti ga na prvo mesto, i tako redom, do kraja niza.

Pozicija minimalnog elementa u nizu počevši od pozicije i:

```
int poz_min(int a[], int n, int i) {
  int m = i, j;
  for (j = i + 1; j < n; j++)
    if (a[j] < a[m])
      m = j;
  return m;
}
```

Selection sort

```
void razmeni(int a[], int i, int j) {
  int tmp = a[i]; a[i] = a[j]; a[j] = tmp;
}

int selectionsort(int a[], int n) {
  int i;
  for (i = 0; i < n - 1; i++)
    razmeni(a, i, poz_min(a, n, i));
}
```

Vremenska složenost je kvadratna, dok je prostorna složenost konstantna (sortiranje u mestu).

Selection sort

Selection sort se može implementirati rekurzivno. Nešto jednostavnija implementacija je ukoliko se umesto dovođenja najmanjeg na početak niza, uradi dovođenje najvećeg na kraj niza. Naravno, pozicija maksimalnog elementa u nizu takođe može da se implementira rekurzivno (ali i ne mora)

```
int poz_max(int a[], int n) {
  if (n == 1)
    return 0;
  else {
    int m = poz_max(a, n-1);
    return a[m] > a[n-1] ? m : n-1;
  }
}
```

Selection sort

Rekurzivna implementacija:

```
void selectionsort(int a[], int n) {
  if (n > 1) {
    razmeni(a, n-1, poz_max(a, n));
    selectionsort(a, n-1);
  }
}
```

Merge sort

Dva već sortirana niza se mogu objediniti u treći sortirani niz samo jednim prolaskom kroz nizove (tj. u linearno vremenu $O(m+n)$ gde su m i n dimenzije polaznih nizova).

```
void merge(int a[], int m, int b[], int n, int c[]) {
  int i, j, k;
  i = 0, j = 0, k = 0;
  while (i < m && j < n)
    c[k++] = a[i] < b[j] ? a[i++] : b[j++];
  while(i < m) c[k++] = a[i++];
  while(j < n) c[k++] = b[j++];
}
```

Merge sort

Merge sort algoritam deli niz na dve polovine (čija se dužina razlikuje najviše za 1), rekurzivno sortira svaku od njih, i zatim objedinjuje sortirane polovine. Izlaz iz rekurzije je jedinični niz.

```
void mergesort_(int a[], int l, int d, int tmp[]) {
  if (l < d) {
    int i, j;
    int n = d - l + 1, s = l + n/2;
    int n1 = n/2, n2 = n - n/2;
    mergesort_(a, l, s-1, tmp);
    mergesort_(a, s, d, tmp);
    merge(a + l, n1, a + s, n2, tmp);
    for (i = l, j = 0; i <= d; i++, j++)
      a[i] = tmp[j];
  }
}
```

Merge sort

- Vremenska složenost — $O(nlogn)$
- Prostorna složenost — $O(n)$

Quick sort

- Osnovna ideja je da se izabere jedan element, tzv pivot i da se on postavi na svoje mesto u nizu i to tako da su svi elementi levo od njega manji od njega, a svi desno od njega veći od njega (korak particionisanja).
- Zatim se postupak rekurzivno ponovi za levi i desni podniz.
- Primer animacije.

Quick sort

```
void qsort_(int a[], int l, int d) {
  if (l < d) {
    /*dovedi pivot na poziciju l*/
    razmeni(a, l, izbor_pivota(a, l, d));

    /*postavi pivot tako da su levo manji od njega,
      a desno veci*/
    int p = particionisanje(a, l, d);

    /*Sortiraj rekurzivno levi i desni podniz*/
    qsort_(a, l, p - 1);
    qsort_(a, p + 1, d);
  }
}
```

Složenost

- Kako bi se dobila jednačina $T(n) = 2T(n/2) + O(n)$ i efikasnost $O(n \log n)$, potrebno je da korak particionisanja (tj. funkcija `particionisanje`) bude izvršen u linearnom vremenu $O(n)$.
- Particionisanje može da se implementira da bude linearno, ali je pravi izbor pivota problem jer nam treba takav da nam podeli niz na dva dela jednakih veličina, a to ne može u konstantnom vremenu.
- Zbog toga je u najgorem slučaju kvadratna složenost.
- U prosečnom slučaju može očekivati relativno ravnomerna raspodela što dovodi do optimalne složenosti ($O(n \log n)$).

Quick sort

- Izbor pivota može da bude različit, najjednostavnije je da se izabere sam levi element. Bolje performanse: npr. za pivot uzme srednji od tri slučajno izabrana elementa niza.
- Particionisanje se takođe može uraditi na različite načine.

```
int particionisanje(int a[], int l, int d) {
  int p = l, j;              /*Pivot je krajnji levi element*/
  for (j = l+1; j <= d; j++)
    if (a[j] < a[l])       /*if(a[j] < pivot)*/
      razmeni(a, ++p, j); /*element koji je manji od pivota
                            dovodimo na pocetak niza*/
  razmeni(a, l, p);        /*dovodimo pivot na njegovu
                            poziciju*/
  return p;                /*vracamo poziciju pivota*/
}
```

Quick sort

- Napomenimo da je *quick sort* algoritam koji u praksi daje najbolje rezultate kod sortiranja dugačkih nizova.
- Međutim, važno je napomenuti da kod sortiranje kraćih nizova naivni algoritmi (npr. *insertion sort*) mogu da se pokažu praktičnijim.
- Većina realnih implementacija *quick sort* algoritma koristi hibridni pristup — izlaz iz rekurzije se vrši kod nizova koji imaju nekoliko desetina elemenata i na njih se primenjuje *insertion sort*.

Pregled

Literatura

- Slajdovi su pripremljeni na osnovu šestog poglavlja knjige Predrag Janičić, Filip Marić: Programiranje 2
- Za pripremu ispita, slajdovi nisu dovoljni, neophodno je učiti iz knjige!