

Algoritmi i strukture podataka - međuispit

23. travnja 2013.

Nije dopušteno korištenje globalnih i statičkih varijabli te naredbe **goto**.

Zadatak 1. (5 bodova)

Jedan zapis datoteke organizirane po načelu raspršenog adresiranja sadrži matični broj studenta (int), ime i prezime (50+1 znak), godinu studija (int) te trenutni prosjek ocjena (float). Prazni se zapis prepoznaje po matičnom broju jednakom nula (0). Veličina bloka na disku je 2048 B. Očekuje se najviše 100000 zapisa, a tablica je dimenzionirana za 35% veći kapacitet. Prilikom upisa primjenjuje se metoda cikličkog preljeva. Ključ zapisa je matični broj, a pretvorba ključa u adresu se obavlja već pripremljenom funkcijom

int adresa(int matbr);

Napisati funkciju koja će odrediti u koji je pretinac izvorno (prilikom upisa u datoteku) bilo upućeno najviše zapisa i potom vratiti broj zapisa upućenih u taj pretinac, a koji su završili u preljevu. Ako ima više takvih pretinaca vratiti rezultat za bilo koji od njih.

Funkcija treba imati prototip:

int fun(FILE *f);

Zadatak 2. (5 bodova)

Napišite rekurzivnu funkciju prototipa:

int prebroji_znamenke(int broj, int znamenka);

koja će odrediti učestalost (broj pojavljivanja) zadane znamenke u zadanom broju.

Napišite glavni program koji od korisnika (putem tipkovnice) prima broj i traženu znamenku, poziva funkciju **prebroji_znamenke** i ispisuje dobiveni rezultat.

Napomena: Nerekurzivno rješenje neće se priznavati.

Zadatak 3. (5 bodova)

Neka je zadano polje **a** koje se sastoji od **n** cijelih brojeva. Napišite rekurzivnu funkciju **maxRekurzivno** prototipa:

int maxRekurzivno(int a[],int n, int *brojPojavljivanja)

koja će pronaći najveći element u polju **a** te njegovu učestalost (koliko puta se taj element pojavljuje u **a**).

Napomena: Nerekurzivno rješenje neće se priznavati.

Zadatak 4. (5 bodova)

a) (4 boda) Napišite funkciju prototipa

```
int trazi_element(int* polje, int brojElemenata, int element)
```

koja u zadanom polju cijelih brojeva *binarnim pretraživanjem* traži određeni element. Funkcija vraća 1 ako polje sadrži traženi element, odnosno -1 u suprotnom. Na raspolaganju je i pomoćna funkcija prototipa:

```
void sortiraj_polje(int* polje, int brojElemenata);
```

koja **silazno** sortira zadano polje.

U glavnom programu dinamički alocirajte polje cijelih brojeva i omogućite unos elemenata polja sve dok korisnik ne unese -1. Nakon toga od korisnika zatražite unos traženog elementa te provjerite i na konzolu ispišite je li uneseni element u prethodno dinamički stvorenom polju cijelih brojeva. Za provjeru pripadnosti elementa polju koristite svoju funkciju **trazi_element**.

b) (1 bod) Kolika je apriorna složenost cijelog postupka pod pretpostavkom da funkcija **sortiraj_polje** primjenjuje **merge sort algoritam**?

Zadatak 5. (5 bodova)

Neka je definiran tip podatka:

```
typedef struct{  
    int prvi;  
    int drugi;  
} podatak;
```

Zadano je polje tipa **podatak** s elementima:

3	7	2	9	3	2	9	4	8	5
4	3	4	5	1	1	6	7	4	0

Ilustrirajte uzlazno sortiranje zadanog polja po varijabli (atributu) **prvi** (ispišite polje nakon svake promjene i označite sve elemente važne za sljedeći korak):

a) (2 boda) algoritmom **mergesort** i

b) (3 boda) algoritmom **quicksort**. Stožer za quicksort birajte metodom aproksimacije medijana temeljem prvog, srednjeg i zadnjeg člana.

*Napomena: Varijabla **drugi** se u ovom zadatku nikad ne koristi kao kriterij sortiranja.*

Rješenja

1.

```
#define N 100000
#define BLOK 2048
#define C BLOK/sizeof (zapis)
#define M (int) (N*1.35/C)

typedef struct z {
    int matbr;
    char ipr[50+1];
    int godina;
    float prosjek;
} zapis;

int fun(FILE *f){
    zapis pretinac[C];
    int i, j, zeljeni_pretinac;
    int max_pretinac = 0, max_vrijednost = 0;
    int broj_zapisa_za_pretinac[M] = {0};
    int broj_preljeva_za_pretinac[M] = {0};
    for (i = 0; i < M; i++) {
        fseek (f, i*BLOK, SEEK_SET);
        fread (pretinac, sizeof (pretinac), 1, f);
        for (j = 0; j < C; j++) {
            if (pretinac[j].sifra != 0) { //provjera je li zapis „pun“
                zeljeni_pretinac = adresa(pretinac[j].sifra);
                broj_zapisa_za_pretinac[zeljeni_pretinac]++;
                if (zeljeni_pretinac != i) {
                    broj_preljeva_za_pretinac[zeljeni_pretinac]++;
                }
            }
            else
                break; //nakon prvog praznog u pretincu, svi su
                //ostali prazni
        }
    }
    for (i = 0; i < M; i++) {
        if (broj_zapisa_za_pretinac[i] > max_vrijednost) {
            max_pretinac = i;
            max_vrijednost = broj_zapisa_za_pretinac[i];
        }
    }
    return broj_preljeva_za_pretinac[max_pretinac];
}
```

Najčešće greške (redoslijed po učestalosti):

- Nepreskakanje u sljedeći pretinac nakon nailaska na prvog praznog u tekućem
- Tretiranje praznog zapisa kao punog
- Krivo shvaćanje zadatka; brojanje zapisa i preljeva u svakom pretincu zasebno i vraćanje pretinca s najviše pripadajućih zapisa u njemu (ili varijacije na temu)

2.

```
int prebroji_znamenke(int broj, int znamenka)
{
    //Negativni broj pretvori u pozitivan
    if (broj < 0) broj *= -1;

    //Sve znamenke do zadnje
    if (broj > 10)
    {
        if (broj % 10 == znamenka) return 1 + prebroji_znamenke(broj/10, znamenka);
        else return prebroji_znamenke(broj/10, znamenka);
    }

    //Zadnja znamenka - uvjet završetka - funkcioniра i za ulaz (0,0)
    else
    {
        if (broj == znamenka) return 1;
        else return 0;
    }
}

int main(int argc, char *argv[])
{
    int broj, znamenka;

    printf("\nUnesite broj: ");
    scanf("%d", &broj);

    printf("\nUnesite znamenku: ");
    scanf("%d", &znamenka);

    printf("\nZnamenka %d pojavljuje se %d puta u broju %d.", znamenka,
        prebroji_znamenke(broj, znamenka), broj);

    return 0;
}
```

Najčešće greške (redoslijed po učestalosti):

- Ne uzimanje u obzir negativnih brojeva
- Loš uvjet terminiranja koji ne provjerava za slučaj ulaza (0,0)
- Krivi rekurzivni poziv
- Neprihvаćanje i neobrađivanje vraćene vrijednosti iz rekurzivnog poziva

3.

```
int maxRekurzivno(int a[],int n, int *brojPojavljivanja)
{
    int kandidat;
    if (n==1){
        *brojPojavljivanja=1;
        return a[0];
    }
    kandidat=maxRekurzivno(a,n-1, brojPojavljivanja);
    if (kandidat>a[n-1]){
        return kandidat;
    }
    else if(kandidat==a[n-1]){
        (*brojPojavljivanja)++;
        return kandidat;
    }
    else{
        *brojPojavljivanja=1;
        return a[n-1];
    }
}
```

Najčešće greške (redoslijed po učestalosti):

- Loš uvjet terminiranja koji vraća 0 što u slučaju polja koji sadrži samo negativne brojeve daje krivo rješenje
- Krivi rad s pokazivačima
- Neprihvatanje i neobrađivanje povratne vrijednosti rekurzivnog poziva
- Nepostavljanje brojača *brojPojavljivanja na 1 u slučaju da smo naišli na veći broj od dosadašnjeg maksimuma
- Rekurzivni poziv izveden na način koji nije sintaksno točan u C-u (!?)
 - Nedostaje ime funkcije, navedene samo zagrade s parametrima
- Mijenjanje sadržaja ulaznog polja
- Pisanje nedohvatljivog koda (npr. iza return naredbe)

4. a)

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

int trazi_element(int* polje, int brojElemenata, int element)
{
    // granice podpolja koje se pretražuje
    int donji, srednji, gornji;

    //sortiranje polja
    sortiraj_polje(polje, brojElemenata);

    // inicijalizacija
    donji = 0; gornji = brojElemenata - 1;

    while (donji <= gornji) {

        // "prepolovi" (pod)polje
        srednji = (donji + gornji) / 2;

        if (element < polje[srednji])
            donji = srednji + 1;          // traženi element u desnom dijelu
        else if (element > polje[srednji])
            gornji = srednji - 1;         // traženi element u lijevom dijelu
        else
            return 1;                    // element pronadjen
    }
    return -1;    // element nije pronadjen
}

int main(int argc, char *argv[])
{
    int *polje = NULL;
    int brojElemenata = 0, element;

    while(1) {
        printf("\nUnesite %d.element polja: ", brojElemenata+1);
        scanf("%d", &element);

        if (element != -1)
        {
            polje = (int *) realloc(polje, (++brojElemenata) * sizeof(int));
            polje[brojElemenata-1] = element;
        }
        else break;
    }

    printf("Unesite elemente koji trazite u polju: ");
    scanf("%d", &element);

    if (trazi_element(polje, brojElemenata, element) == 1)
    {
        printf("Zadani element se nalazi u polju.");
    }
    else
    {
        printf("Zadani element se ne nalazi u polju.");
    }
    return 0;
}
```

Postoji više mogućih realizacija funkcije `trazi_element` koje se mogu dosta razlikovati u izgledu koda. Studenti su češće pisali rekursivnu inačicu funkcije `trazi_element` što je, naravno, posve prihvatljivo.

Najčešće greške (redoslijed po učestalosti):

- Nema obrazloženja u b) dijelu zadatka
- Loše riješen unos polja
 - Korišten `malloc` umjesto `realloc` u petlji (ne čuvaju se stari podatci)
 - Korištenje polja fiksne veličine alociranog na stogu
- Poziv funkcije `sortiraj_polje` na način koji nije sintaksno točan u C-u (!)
- Krivi odabir dijela polja za sljedeću iteraciju
- Nepoznavanje binarnog pretraživanja - korištenje slijednog pretraživanja
- Mijenjanje sadržaja ulaznog polja u `trazi_element`

b)

Složenost merge sort algoritma: **$O(n \cdot \log n)$**

Složenost binarnog pretraživanja: **$O(\log_2 n)$**

*Složenost unosa polja: **$O(n)$** (nenavođenje ove komponente u obrazloženju se toleriralo zbog nejednoznačnosti teksta podzadatka b)*

Budući da se u okviru svakog poziva funkcije trazi_element polje najprije sortira merge sort algoritmom (pozivom funkcije sortiraj_polje), ukupna složenost je: **$O(n \cdot \log n) + O(\log_2 n)$** , tj. **$O(n \cdot \log n)$**

5. **Mergesort** – uzlazno. Više je mogućih, točnih, rješenja - ovisno o izborima. Ovdje je dano rješenje za konzistentno veću lijevu podparticiju u slučaju dijeljenja particije neparne veličine, te u slučaju jednakosti ključeva uzimanje elementa iz lijeve particije (ovaj zadnji izbor osigurava stabilnost algoritma). Zadatak za mergesort je moguće riješiti i sortiranjem samo niza vrijednosti u varijabli **prvi** i nadopunjavanje dobivenog niza na kraju dodajući adekvatno poredane vrijednosti varijabli **drugi** koristeći svojstvo stabilnosti mergesorta (za quicksort u sljedećem zadatku to ne možete).

3	7	2	9	3	2	9	4	8	5
4	3	4	5	1	1	6	7	4	0

2	3	7	9	3	2	9	4	8	5
4	4	3	5	1	1	6	7	4	0

2	3	7	3	9	2	9	4	8	5
4	4	3	1	5	1	6	7	4	0

2	3	3	7	9	2	9	4	8	5
4	4	1	3	5	1	6	7	4	0

2	3	3	7	9	2	4	9	8	5
4	4	1	3	5	1	7	6	4	0

2	3	3	7	9	2	4	9	5	8
4	4	1	3	5	1	7	6	0	4

2	3	3	7	9	2	4	5	8	9
4	4	1	3	5	1	7	0	4	6

2	2	3	3	4	5	7	8	9	9
4	1	4	1	7	0	3	4	5	6

Najčešće greške (redoslijed po učestalosti):

- Sortiranje samo niza vrijednosti u varijabli **prvi** bez ikad navođenja pripadnih vrijednosti **drugi**; u zadatku je dano polje tipa *podatak* koji je strukturni tip i varijable **prvi** i **drugi** su nerazdvojive, sortiranje smije mijenjati samo redoslijed, ali ne i originalne podatke članova polja. Uz to, vrijednosti varijable **drugi** su važne i ne mogu se ignorirati jer nam omogućuju da razlikujemo podatke s identičnim vrijednostima varijable **prvi**.
- Nekonzistentnost u izvođenju algoritma (pogotovo u pogledu korištenih izbora)
- Sortiranje samo niza vrijednosti varijable **prvi** te na kraju, u zadnjem dijelu rješenja adekvatno poredane vrijednosti varijable **drugi**, ali nije se navelo da se pritom koristi svojstvo stabilnosti mergesorta

Quicksort – uzlazno. Više je mogućih, točnih, rješenja - ovisno o izborima. Ovdje je dano rješenje za konzistentno biranje stožera lijevo u slučaju nepostojanja jedinstvenog srednjeg indeksa i prebacivanje jednakih elemenata u lijevu particiju. Također je napravljen izbor kod insertion sorta, korištenog za sortiranje particija s manje od 3 elementa, da redosljed elemenata s istim ključevima sortiranja(prvi) ostaje isti.

3	7	2	9	3	2	9	4	8	5
4	3	4	5	1	1	6	7	4	0

3	7	2	9	3	2	9	4	8	5
4	3	4	5	1	1	6	7	4	0

3	7	2	9	8	2	9	4	3	5
4	3	4	5	4	1	6	7	1	0

3	2	2	9	8	7	9	4	3	5
4	1	4	5	4	3	6	7	1	0

3	2	2	3	8	7	9	4	9	5
4	1	4	1	4	3	6	7	5	0

2	2	3	3	8	7	9	4	9	5
1	4	4	1	4	3	6	7	5	0

2	2	3	3	5	7	8	4	9	9
1	4	4	1	0	3	4	7	5	6

2	2	3	3	5	7	9	4	8	9
1	4	4	1	0	3	5	7	4	6

Najčešće greške (redosljed po učestalosti):

- Sortiranje samo niza vrijednosti u varijabli *prvi* bez ikad navođenja pripadnih vrijednosti *drugi*; u zadatku je dano polje tipa *podatak* koji je strukturni tip i varijable *prvi* i *drugi* su nerazdvojive, sortiranje smije mijenjati samo redosljed, ali ne i originalne podatke članova polja. Uz to, vrijednosti varijable *drugi* su važne i ne mogu se ignorirati jer nam omogućuju da razlikujemo podatke s identičnim vrijednostima varijable *prvi*.
- Nekonzistentnost u izvođenju algoritma (pogotovo u pogledu korištenih izbora)
- Preskakanje koraka
- Krivo izvedeno rukovanje stožerom
 - Uopće nema skrivanja stožera
 - Kod skrivanja i vraćanja stožera svi se elementi između pomiču lijevo ili desno, umjesto samo zamjene stožera s nekim drugim elementom
- Krivo preslagivanje elemenata u particije

2	2	3	3	5	7	4	9	8	9
1	4	4	1	0	3	7	5	4	6

2	2	3	3	5	7	4	8	9	9
1	4	4	1	0	3	7	4	5	6

2	2	3	3	4	5	7	8	9	9
1	4	4	1	7	0	3	4	5	6