

Programiranje 1

12. predavanje

Saša Singer

`singer@math.hr`

`web.math.pmf.unizg.hr/~singer`

PMF – Matematički odsjek, Zagreb

Sadržaj predavanja

- Složene strukture podataka: nizovi (polja):
 - Definicija jednodimenzionalnog polja.
 - Inicijalizacija jednodimenzionalnog polja.
 - Polje kao argument funkcije.
 - Pokazivači i jednodimenzionalna polja.
- Osnovne operacije s nizovima podataka (poljima):
 - Zbrajanje članova niza.
 - Najmanji (najveći) element u nizu.
- Pretraživanje i sortiranje nizova (polja):
 - Sekvencijalno pretraživanje.
 - Binarno pretraživanje sortiranog niza.

Informacije — rezultati PK1

Praktični kolokvij, prvi krug PK1 — prošlo vas je 123 od 211.

● Rezultati su “osrednji” — oko 58.29% svih došlih.

Statistika za nekoliko ranijih godina:

godina	popis	PK1	došlo	prošlo	palo	% prošlih
2019	244	224	211	123	88	58.29%
2018	258	237	224	153	71	68.30%
2017	265	257	249	151	98	60.64%
2016	276	266	253	153	100	60.74%
2015	291	276	265	175	90	66.04%
2014	272	257	252	149	103	59.13%
2012	279	263	252	146	106	57.94%

Praktični kolokvij — popravak (*Ne koristiti!*)

Broj studenata za **popravak** (PK2):

- Pravo na **popravak** PK2 (bez molbe) ima 88 studenata,
- i (potencijalno) još 2 studenta s molbom.

Drugi krug PK2 ide

- drugi **puni** tjedan nastave **iza** praznika,
- ponedjeljak, 13. 1. — petak, 17. 1.,
- svi termini su poznati.

Prijava za PK2 = “zauzimanje termina” je **tjedan ranije**,

- od **utorka**, 7. 1. — tj. već ide (i još traje),
- na **papirima** desno od mog ureda (pišite **čitljivo**)!

Praktični kolokvij — nedolazak

Nedolazak na PK = izostanak s kolokvija!

Za naknadno polaganje, izostanak treba opravdati i to

- urudžbiranom molbom za naknadno polaganje, zajedno s ispričnicom.

U protivnom, nema praktičnog i nemate pravo na popravak!

Tj. preduvjet za popravak praktičnog je pojavljivanje (i pad) na praktičnom.

Informacije

Zadnje predavanje u ovom semestru je:

- drugi petak iza praznika, 17. 1. 2020. godine.

I isplati se doći, radimo

- sortiranje nizova i završne primjere za kolokvij.

Zadnji petak, 24. 1., sam isto “tu” od 10 sati

- i možete me “pitati” (u uredu sam, ili ispred faksa).

Konzultacije od 12 sati, također, “rade” — slobodno dođete!

Informacije — dogovor za usmeni, ocjene

Za eventualni **usmeni** (po želji/“kazni”):

- 📌 pogledati **obavijest** na **webu**, uz **rezultate** kolokvija!

Uobičajeno,

- 📌 **dogovor za usmeni** je u vrijeme **uvida** (ne kasnije), a **usmeni** je nekoliko dana kasnije.

Više **nema** indeksa — pa **nema** ni **upisa ocjena**.

- 📌 Ocjene šaljemo na upis u **ISVU** (studomat) nekoliko dana **nakon** uvida, a najkasnije dan nakon usmenih.

Nizovi podataka (jednodimenzionalna polja)

Sadržaj

- Složene strukture podataka: nizovi (polja):
 - Definicija jednodimenzionalnog polja.
 - Inicijalizacija jednodimenzionalnog polja.
 - Polje kao argument funkcije.
 - Pokazivači i jednodimenzionalna polja.

Polje

Polje je konačan niz varijabli istog tipa, sa zajedničkim imenom, numeriranih nenegativnim cjelobrojnim indeksima.

U C-u — indeks uvijek počinje od nule.

Polje je vrlo slično vektoru u matematici: $x = (x_1, \dots, x_n)$, samo se indeksi broje od nule, pa vektor x s n komponenti u C-u ima oblik: $x = (x_0, \dots, x_{n-1})$.

Primjer.

```
double x[3]; /* polje x s 3 clana tipa double */
x[0] = 0.2;
x[1] = 0.7;
x[2] = 5.5;
/* x[3] = 4.4;    -> greska, x[3] nije definiran! */
```

Definicija jednodimenzionalnog polja

Jednodimenzionalno polje **definira** se na sljedeći način:

```
mem_klasa tip ime[izraz];
```

gdje je:

- **mem_klasa** = **memorijska klasa** cijelog polja (v. **Prog2**),
- **tip** = **tip podatka** svakog **elementa** polja,
- **ime** = **ime polja** = zajednički dio imena svih elemenata,
 - ujedno = **adresa prvog** elementa polja, tj. **&ime[0]**,
- **izraz** = konstantan, cjelobrojni, **pozitivan** izraz — koji zadaje **broj** elemenata u polju (duljina polja). Najčešće je
 - **pozitivna** cjelobrojna ili simbolička **konstanta**.

Definicija jednodimenzionalnog polja (nastavak)

Elementi jednodimenzionalnog polja su:

`ime[0], ..., ime[izraz - 1]`.

Svaki element `ime[i]` je “obična” varijabla tipa `tip`.

Deklaracija memorijske klase nije obavezna.

Polje deklarirano bez memorijske klase:

- unutar funkcije je automatska varijabla (rezervacija memorije na “run-time stacku”, ulaskom u funkciju),
- a izvan svih funkcija je statička varijabla.

Unutar neke funkcije, polje se može učiniti statičkim pomoću identifikatora memorijske klase `static` (v. Prog2).

Ključne stvari — za bilo koje polje u C-u

Zapamtiti: Ime polja je sinonim za

- konstantni pokazivač na prvi element polja
- = adresa prvog elementa polja (više u Prog2).

Radi efikasnosti pristupa, elementi polja smještaju se u

- uzastopne memorijske lokacije — redom po indeksu.

Dakle, polje u memoriji jednoznačno je definirano

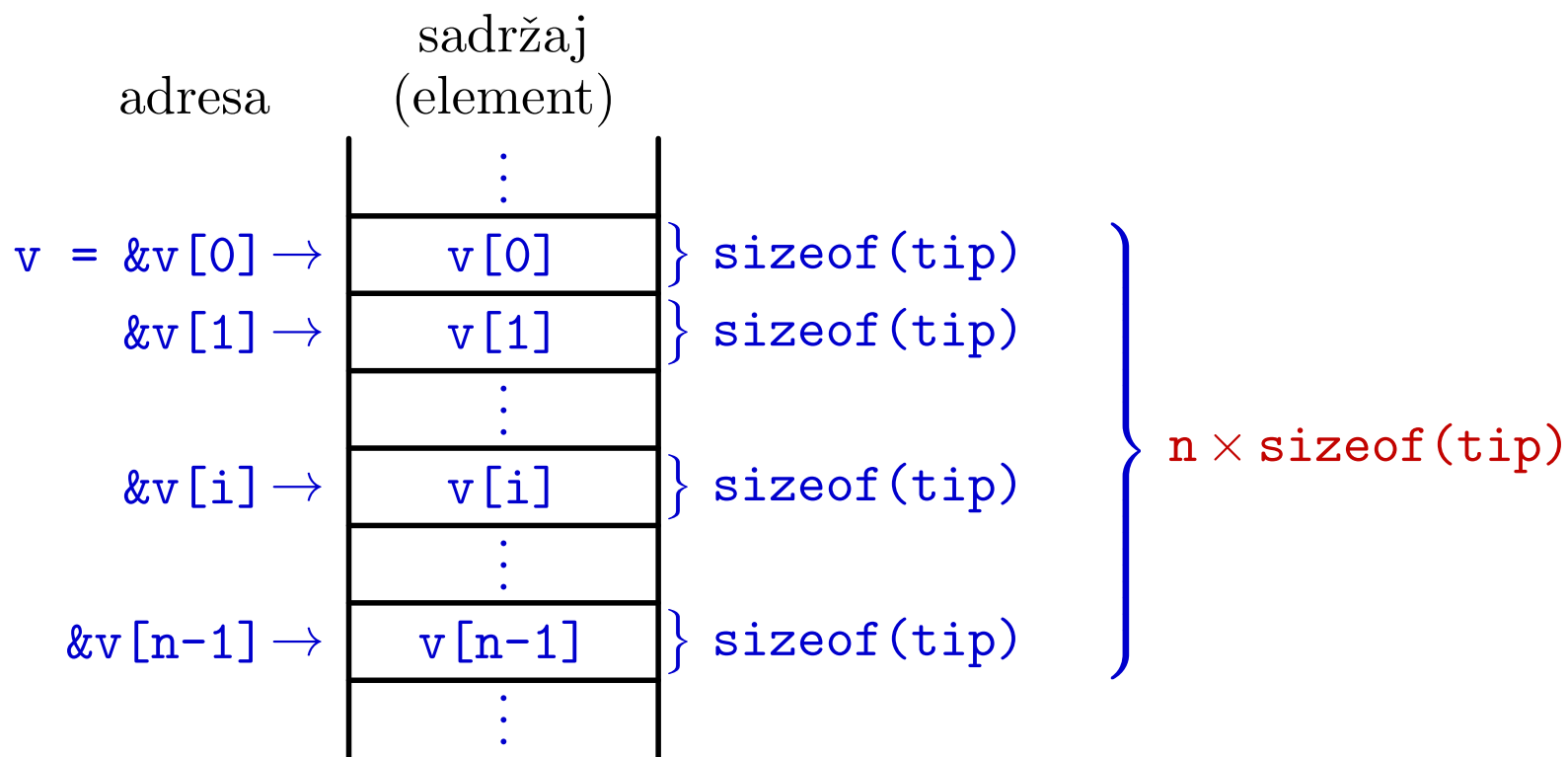
- početnom adresom polja = ime polja,
- tipom svakog elementa = duljina elementa, i
- brojem elemenata!

Adresa svakog elementa se onda “zna”, tj. može se izračunati,

- jer su elementi polja spremljeni “u bloku”.

Spremanje polja u memoriji i adrese elemenata

Nakon definicije `tip v[n];` polje `v` izgleda ovako u memoriji:



Adresa `i`-tog elementa je (matematički, a ne C-aški pisano):

$$\&v[i] = v + i * \text{sizeof}(\text{tip}), \text{ za } i = 0, \dots, n-1.$$

Oprez — nema kontrole granica za indekse

Vidimo da **adresa** elementa $v[i]$ ovisi **samo** o

- **početnoj adresi** polja = **ime** polja,
- **tipu** elementa = **duljina** pojedinog elementa, i
- **indeksu** elementa!

Broj elemenata u polju (iz **definicije** polja) za to **nije bitan**.

- On služi samo pri **inicijalnoj rezervaciji** memorije za polje, i nigdje se posebno **ne pamti**.

Zato, u **C-u** **nema kontrole** granica za **indeks** elementa polja.

- Programer **mora** voditi računa o tome da **ne gazi** po memoriji — **izvan** dozvoljenih (rezerviranih) granica!

Samo u “**trivijalnim**” slučajevima prevoditelj **može** kontrolirati indekse — i to tamo gdje je **rezervirana** memorija za polje.

Inicijalizacija polja

Polja se mogu **inicijalizirati** — element, po element,

- navođenjem popisa **vrijednosti** elemenata unutar **vitičastih** zagrada.
- U tom popisu, pojedine vrijednosti **odvojene** su **zarezom** (koji **nije** operator).

Sintaksa:

```
mem_klasa tip ime[izraz] = {v_1, ..., v_n};
```

što daje

```
ime[0] = v_1, ..., ime[n - 1] = v_n.
```


Inicijalizacija polja (nastavak)

Primjer.

```
double v[3] = {1.17, 2.43, 6.11};
```

je ekvivalentno s

```
double v[3];  
v[0] = 1.17;  
v[1] = 2.43;  
v[2] = 6.11;
```

Inicijalizacija polja (nastavak)

Ako je broj inicijalizacijskih vrijednosti n

- veći od duljine polja — javlja se greška,
- manji od duljine polja, onda će preostale vrijednosti biti inicijalizirane nulom.

Prilikom inicijalizacije, duljina polja ne mora biti zadana.

- Tada se duljina polja računa automatski, iz broja inicijalizacijskih vrijednosti.

Primjer. Možemo pisati

```
double v[] = {1.17, 2.43, 6.11};
```

što kreira polje v duljine 3 elementa, i inicijalizira ga.

Inicijalizacija polja (nastavak)

Polja znakova mogu se inicijalizirati znakovnim nizovima.

Primjer. Naredbom

```
char c[] = "tri";
```

definirano je polje od 4 znaka:

```
c[0] = 't', c[1] = 'r', c[2] = 'i', c[3] = '\0'.
```

Takav način pridruživanja dozvoljen je samo u definiciji varijable (kao inicijalizacija). Nije dozvoljeno pisati:

```
c = "tri"; /* Pogresno! Koristiti strcpy! */
```

jer lijeva strana pridruživanja ne smije biti polje (ime polja je konstantni pokazivač — adresa prvog elementa).

Primjer — aritmetička sredina

Primjer. Računanje aritmetičke sredine.

```
#include <stdio.h>
int main(void) {
    double v[] = {2.0, 3.11, 4.05, -1.07};
    int n = sizeof(v) / sizeof(double), i;
    double a_sredina = 0.0;

    for(i = 0; i < n; ++i)
        a_sredina += v[i];
    a_sredina /= n;
    printf(" Sredina je %20.12f\n", a_sredina);
    return 0;
}
```

Polje kao argument funkcije

Polje može biti formalni i stvarni argument funkcije. U tom slučaju:

- ne prenosi se cijelo polje po vrijednosti (kopija polja!),
- već funkcija dobiva (po vrijednosti) pokazivač na neki element polja.

Taj pokazivač funkcija (lokalno) “vidi” kao

- pokazivač na prvi “radni” element polja — iako,
- stvarno, on ne mora biti adresa prvog elementa u polju.

Unutar funkcije, elementi polja mogu se

- dohvatiti i promijeniti, korištenjem indeksa polja.

Razlog: tzv. aritmetika pokazivača (v. drugi semestar).

Polje kao argument funkcije (nastavak)

Funkciju **f**, koja prima **polje v** s elementima tipa **tip** kao argument, možemo deklarirati na **dva** načina:

```
f(typ v[], ...)    ili    f(typ *v, ...)
```

U **prvom** načinu **ne treba** navesti duljinu polja. **Drugi** način direktno kaže da je ime polja **v pokazivač** na objekt tipa **tip** i podrazumijeva se da je to **adresa prvog elementa polja**.

Ako **ne želimo** da funkcija **mijenja** elemente polja **unutar** funkcije, onda **dodajemo** ključnu riječ **const** na početku deklaracije argumenta (na pr., prvi string u **scanf**, **printf**):

```
f(const typ v[], ...)    ili    f(const typ *v, ...)
```

Polje kao argument funkcije (nastavak)

Primjer. Funkciju, koja prima **polje** realnih brojeva (tipa **double**) i računa **srednju vrijednost svih** elemenata polja, možemo napisati ovako:

```
double srednja_vrijednost(int n, const double v[]) {
    int i;
    double suma = 0.0;

    for (i = 0; i < n; ++i) suma += v[i];
    return suma / n;
}
```

Uočite da je **broj** elemenata **n**, također, **argument** funkcije. Inače, funkcija **ne zna broj** “radnih” elemenata (osim iz neke globalne varijable).

Polje kao argument funkcije (nastavak)

Pri **pozivu** funkcije, koja ima **polje** kao **formalni** argument, **stvarni** argument je

- ime polja ili pokazivač na “**prvi radni**” element u polju.

```
int main(void) {
    int n;
    double v[] = {1.0, 2.0, 3.0}, sv;

    n = 3;
    sv = srednja_vrijednost(n, v);
    return 0;
}
```

Poziv `srednja_vrijednost(2, &v[1])` je **korektan** (v. iza)!

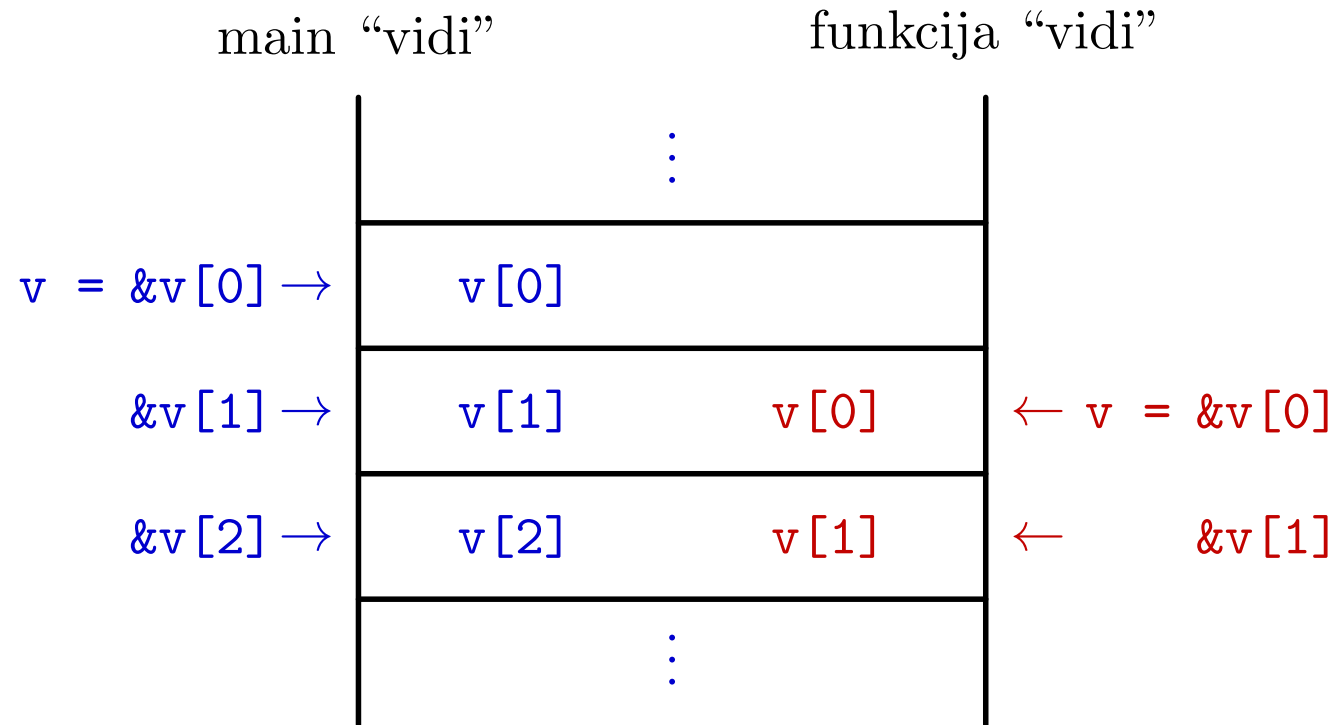
Polje v u glavnom programu i u funkciji

Poziv `srednja_vrijednost(2, &v[1])` radi sljedeće:

• lokalna varijabla v u funkciji poprima vrijednost

$$v = \&v[1], \quad \text{za } v \text{ iz main.}$$

Iza toga,



Aritmetika pokazivača — ukratko

Već smo rekli: **Ime polja** je

- **konstantni pokazivač** na **prvi element** u polju.

Ako je **a** neko polje, onda je: $a = \&a[0]$ ili $*a = a[0]$.

Vrijedi i “**obrat**”: Svaki **pokazivač** na neki objekt možemo interpretirati i kao

- **pokazivač** na **prvi element** u **polju** objekata tog tipa.

Na primjer, tako koristimo vezu **pokazivač** — **polje** u **funkciji**.

Elementi polja spremaju se na **uzastopnim** lokacijama u memoriji. Zato, za **svaki element** polja **a**, vrijedi veza:

- $a + i = \&a[i]$ ili $*(a + i) = a[i]$, za svaki **i**.

Stvarne adrese ovise o “**duljini**” elemenata u polju, tj. o **tipu**.

Pokazivači i jednodimenzionalna polja

Ime **bilo kojeg** polja, pa onda i **jednodimenzionalnog** polja je

● **konstantni pokazivač** na **prvi** element polja!

Primjer.

```
int a[10], b[10];  
...  
a = a + 1; /* Greska, a je konst. pokazivac. */  
b = a;     /* Greska! */
```

Napomena. Ta **adresa** prvog elementa polja **nije** spremljena u neku memorijsku lokaciju (kao varijabla) i zato se **ne smije** mijenjati.

Prevoditelj ju “**pamti**” kad **rezervira** memoriju za cijelo polje, a zatim, “**vodi računa**” o **adresama** — preko **indeksa**.

Pokazivači i jednodimenzionalna polja (nast.)

Primjer.

```
int a[10], *pa;
...
pa = a;      /* ekviv. s pa = &a[0]; */
pa = pa + 2; /* Nije greska - &a[2] */
pa++;       /* &a[3] */
```

Primjer.

```
int a[10], *pa;
...
pa = &a[0];
*(pa + 3) = 20; /* ekviv. s a[3] = 20; */
*(a + 1) = 10; /* ekviv. s a[1] = 10; */
```

Prioriteti i asocijativnost

Primjer. Važnost prioriteta i asocijativnosti. Neka je

```
int a[4] = {0, 10, 20, 30};  
int *ptr, x;  
ptr = a;
```

Nakon izvršavanja navedenih naredbi (**tim redom**), dobivamo

naredba	x	ptr	a[0]	a[1]	a[2]	a[3]
x = *ptr;	0	0098F830	0	10	20	30
x = *ptr++;	0	0098F834	0	10	20	30
x = (*ptr)++;	10	0098F834	0	11	20	30
x = *++ptr;	20	0098F838	0	11	20	30
x = ++(*ptr);	21	0098F838	0	11	21	30

Važnost prioriteta i asocijativnosti

Objašnjenje. Unarni operatori `&`, `*`, `++` i `--` imaju **viši** prioritet od **aritmetičkih** operatora i operatora **pridruživanja**.

```
*ptr += 1;    /* ili samo izraz *ptr + 1 */
```

Prvo djeluje `*`. Zato se **povećava** za **jedan**

👉 **vrijednost** na koju **ptr** pokazuje (***ptr**), a **ne** pokazivač.

Zbog **asocijativnosti unarnih** operatora $D \rightarrow L$, isti izraz možemo napisati kao

```
++*ptr    /* povecava *ptr */
```

(prvo **dereferenciranje**, pa **inkrementiranje**, pa iskoristi **povećanu** vrijednost ***ptr**).

Važnost prioriteta i asocijativnosti (nastavak)

Kod **postfiks** notacije operatora **inkrementiranja**,

- ako želimo **povećati** ili **smanjiti sadržaj**, moramo koristiti **zagrade**.

```
(*ptr)++    /* povecava *ptr */
```

Izraz bez zagrada

```
*ptr++    /* povecava pokazivac ptr */
```

inkrementira **pokazivač ptr**, i to **nakon** što iskoristi ***ptr** (**vrijednost** na koju **ptr** pokazuje).

Osnovne operacije s nizovima

Sadržaj

- Osnovne operacije s nizovima podataka (poljima):
 - Zbrajanje članova niza.
 - Najmanji (najveći) element u nizu.

Zbroj svih članova niza

Zadan je niz (polje) od n realnih brojeva

$$x_0, x_1, \dots, x_{n-1}.$$

Treba naći **zbroj** svih članova niza. Pretpostavka je $n > 0$.

Algoritam: (recimo da su x_i tipa **double**)

```
...
zbroj = 0.0;
for (i = 0; i < n; ++i)
    zbroj += x[i];
...
printf("Zbroj članova niza = %f.\n", zbroj);
```

Ovo radi za bilo koji n (može i $n \leq 0$), uz **dogovor** **zbroj = 0**.

Zbroj svih članova niza (nastavak)

Funkcija za zbrajanje:

```
double zbroj_clanova(int n, double x[])
{
    int i;
    double zbroj = 0.0;

    for (i = 0; i < n; ++i)
        zbroj += x[i];
    return zbroj;
}
```

Zbroj svih članova niza (nastavak)

Poziv funkcije:

```
int main(void) {
    int n = 5;
    double v[] = {1.2, 2.6, 1.8, 4.4, 0.8};

    printf("Zbroj svih članova niza = %f\n",
           zbroj_clanova(n, v) );

    printf("Zbroj srednja tri člana niza = %f\n",
           zbroj_clanova(3, &v[1]) );

    return 0;
}
```

Najmanji član niza

Tražimo najmanji član niza od n realnih brojeva

$$x_0, x_1, \dots, x_{n-1}.$$

Pretpostavka je opet $n > 0$. Ovdje se “tvrdo” koristi za korektnu inicijalizaciju — neprazan niz.

Najmanji član niza (nastavak)

Algoritam: vrijednost i indeks (pozicija) najmanjeg elementa

```
min = x[0];
poz = 0;

for (i = 1; i < n; ++i)
    if (x[i] < min) {
        min = x[i];
        poz = i;
    }

...
printf("Najmanji član niza: x[%d] = %f\n",
      poz, min);
```

Složenost: $n - 1$ usporedbi članova niza.

Najmanji član niza (nastavak)

Funkcija koja vraća samo vrijednost najmanjeg elementa:

```
double min_clan(int n, double x[])
{
    int i;
    double min = x[0];

    for (i = 1; i < n; ++i)
        if (x[i] < min)
            min = x[i];
    return min;
}
```

Sami: Funkcija koja vraća i indeks (poziciju) najmanjeg elementa, kao varijabilni argument.

Pretraživanje nizova

Sadržaj

- Pretraživanje nizova (polja):
 - Sekvencijalno pretraživanje.
 - Složenost sekvencijalnog pretraživanja.
 - Binarno pretraživanje sortiranog niza.
 - Složenost binarnog pretraživanja.

Problem pretraživanja nizova

Problem **pretraživanja** — opća formulacija:

- Treba **provjeriti** **nalazi** li se zadani element **elt** među članovima zadanog niza

$$x_0, x_1, \dots, x_{n-1}.$$

Drugim riječima, **pitanje** glasi:

- **postoji** li indeks $i \in \{0, \dots, n-1\}$ takav da je $\text{elt} = x_i$.

Za početak, želimo samo **odgovor** na ovo pitanje, tj. rezultat pretrage je

- logička vrijednost **nasli** — **1 (istina)** ili **0 (laž)**.

Sekvencijalno pretraživanje

Ako niz **nije** sortiran, tj. u nizu vlada “**nered**”, koristimo

- **sekvencijalno** pretraživanje (“jedan po jedan”).

Pretraživanje se vrši **sve dok** su ispunjena **2 uvjeta**:

- **nismo našli** traženi element, **i**
- dok se indeks **i** nalazi **unutar** dozvoljenih granica, a te granice su — od **0** do **$n - 1$** .

Očito, potraga je završena (u **najgorem** slučaju = **nema** ga)

- nakon točno **jednog** prolaza kroz **sve** elemente.

Ona može završiti i **prije**, ako se traženi element **nalazi** negdje **prije kraja** niza — recimo, na **početku** niza.

Sekvencijalno pretraživanje — algoritam

Algoritam:

```
nasli = 0;
i = 0;

while (!nasli && i < n) {
    if (x[i] == elt)
        nasli = 1;
    else
        ++i;
}
```

Napomena. Prvi uvjet `!nasli` može se **ispustiti**, ako koristimo `break` kad **nađemo** element. Napišite sami!

Sekvencijalno pretraživanje — primjer 1

Primjer. U polju od 7 elemenata ispitajte nalazi li se broj 55.

42	12	55	94	18	44	67
----	----	----	----	----	----	----

Sekvencijalno pretraživanje — primjer 1

Primjer. U polju od 7 elemenata ispitajte nalazi li se broj 55.

42	12	55	94	18	44	67
----	----	----	----	----	----	----



`i = 0`

`x[0] != 55`

`nasli = 0`

Sekvencijalno pretraživanje — primjer 1

Primjer. U polju od 7 elemenata ispitajte nalazi li se broj 55.

42	12	55	94	18	44	67
----	----	----	----	----	----	----



`i = 1`

`x[1] != 55`

`nasli = 0`

Sekvencijalno pretraživanje — primjer 1

Primjer. U polju od 7 elemenata ispitajte nalazi li se broj 55.

42	12	55	94	18	44	67
----	----	----	----	----	----	----



`i = 2`

`x[2] == 55`

`nasli = 1`

Sekvencijalno pretraživanje — primjer 2

Primjer. U polju od 7 elemenata ispitajte nalazi li se broj 21.

42	12	55	94	18	44	67
----	----	----	----	----	----	----

Sekvencijalno pretraživanje — primjer 2

Primjer. U polju od 7 elemenata ispitajte nalazi li se broj 21.

42	12	55	94	18	44	67
----	----	----	----	----	----	----



`i = 0`

`x[0] != 21`

`nasli = 0`

Sekvencijalno pretraživanje — primjer 2

Primjer. U polju od 7 elemenata ispitajte nalazi li se broj 21.

42	12	55	94	18	44	67
----	----	----	----	----	----	----



```
i = 1
```

```
x[1] != 21
```

```
nasli = 0
```

Sekvencijalno pretraživanje — primjer 2

Primjer. U polju od 7 elemenata ispitajte nalazi li se broj 21.

42	12	55	94	18	44	67
----	----	----	----	----	----	----



`i = 2`

`x[2] != 21`

`nasli = 0`

Sekvencijalno pretraživanje — primjer 2

Primjer. U polju od 7 elemenata ispitate nalazi li se broj 21.

42	12	55	94	18	44	67
----	----	----	----	----	----	----



`i = 3`

`x[3] != 21`

`nasli = 0`

Sekvencijalno pretraživanje — primjer 2

Primjer. U polju od 7 elemenata ispitajte nalazi li se broj 21.

42	12	55	94	18	44	67
----	----	----	----	----	----	----



`i = 4`

`x[4] != 21`

`nasli = 0`

Sekvencijalno pretraživanje — primjer 2

Primjer. U polju od 7 elemenata ispitajte nalazi li se broj 21.

42	12	55	94	18	44	67
----	----	----	----	----	----	----



`i = 5`

`x[5] != 21`

`nasli = 0`

Sekvencijalno pretraživanje — primjer 2

Primjer. U polju od 7 elemenata ispitajte nalazi li se broj 21.

42	12	55	94	18	44	67
----	----	----	----	----	----	----



`i = 6`

`x[6] != 21`

`nasli = 0`

Sekvencijalno pretraživanje — primjer 2

Primjer. U polju od 7 elemenata ispitajte nalazi li se broj 21.

42	12	55	94	18	44	67
----	----	----	----	----	----	----

Sekvencijalno pretraživanje — funkcija

Funkcija koja vraća odgovor:

```
int seq_search(int x[], int n, int elt)
{
    int i;

    for (i = 0; i < n; ++i)
        if (x[i] == elt)
            return 1;

    return 0;
}
```

Koristimo “skraćenu” pretragu, bez varijable *nasli*.

Sekvencijalno pretraživanje — složenost

Složenost pretraživanja mjerimo brojem usporedbi

• “jednak”, odnosno, “različit” (jer nema uređaja),

i to samo u tipu za članove niza. Operacije na indeksima ne brojimo — njih ima podjednako mnogo kao i usporedbi.

U najgorem slučaju, moramo provjeriti sve članove niza, tj.

$$\text{broj usporedbi} \leq n.$$

Ova mjera složenosti je dobra procjena za trajanje izvršavanja algoritma sekvencijalnog pretraživanja — oznaka $T(n)$.

Zapis za trajanje:

$$T(n) \in O(n).$$

Značenje: trajanje, u najgorem slučaju, linearno ovisi o n .

Točno značenje zapisa složenosti

Točno matematičko značenje zapisa

$$T(n) \in O(f(n))$$

za neke funkcije T i f (sa skupa \mathbb{N} u skup \mathbb{R}):

Postoji konstanta $c \in \mathbb{R}$ i postoji $n_0 \in \mathbb{N}$, takvi da, za svaki $n \in \mathbb{N}$, vrijedi implikacija

$$n \geq n_0 \implies T(n) \leq c \cdot f(n).$$

Prijevod: T raste sporije od “neka konstanta puta f ”, za sve dovoljno velike n .

Napomena. Često se piše $T(n) = O(f(n))$, što nije korektno, jer ova “jednakost” nije simetrična!

Binarno pretraživanje

Ako je niz **uzlazno** ili **silazno sortiran**, tj. vrijedi

$$x_0 \leq x_1 \leq \dots \leq x_{n-1} \quad \text{ili} \quad x_0 \geq x_1 \geq \dots \geq x_{n-1},$$

potraga se može drastično **ubrzati**, tako da koristimo tzv.

📍 **binarno** pretraživanje — pretraživanje “**raspolavljanjem**”.

Zamislite potragu (po prezimenu) u telefonskom imeniku velegrada. Kako bismo to proveli?

📍 Otvorili bismo imenik na **nekom** mjestu. Ako je traženo prezime **ispred** prezimena na otvorenom mjestu, onda bismo postupak ponovili s **prvim** dijelom imenika, a ako je **iza**, onda s **drugim** dijelom imenika.

Pitanje je — **gdje** je to “**neko**” mjesto?

Binarno pretraživanje (nastavak)

Vratimo se na apstraktni model. Za elemente niza vrijedi

$$x_0 \leq x_1 \leq \dots \leq x_i \leq \dots \leq x_{n-2} \leq x_{n-1},$$

pri čemu je x_i odabrani objekt, kojeg ćemo usporediti sa zadanim elementom `elt`.

Kako ne znamo koji su elementi u nizu,

🔴 niz je najbolje podijeliti “na pola”.

Onda je podjednako vjerojatno da se `elt` nalazi u prvom ili drugom dijelu — jer su prvi i drugi dio podjednake veličine.

U tom slučaju, bez obzira gdje se element nalazi, potragu smo

🔴 smanjili na podniz s polovičnim brojem elemenata.

Binarno pretraživanje (nastavak)

Precizno, neka je $l = 0$ indeks prvog, a $d = n - 1$ indeks zadnjeg elementa u nizu. Srednji element i ima indeks

$$i = \left\lfloor \frac{l + d}{2} \right\rfloor \quad \text{ili} \quad i = \left\lceil \frac{l + d}{2} \right\rceil.$$

Budući da cjelobrojnim dijeljenjem u C-u dobijemo prvi izbor, onda se, obično, on koristi kao “sredina”.

Elemente niza x svrstali smo u 3 skupine:

- elementi s indeksima od $l = 0$ do $i - 1$,
- element s indeksom i ,
- elementi s indeksima od $i + 1$ do $d = n - 1$.

Binarno pretraživanje (nastavak)

Postavljamo 3 pitanja:

- $\text{elt} < x_i?$ Odgovor “da” znači da nastavljamo tražiti
 - u podnizu s indeksima od l do $d = i - 1$ (ispred x_i),
- $\text{elt} > x_i?$ Odgovor “da” znači da nastavljamo tražiti
 - u podnizu s indeksima od $l = i + 1$ do d (iza x_i),
- $\text{elt} = x_i?$ Odgovor “da” znači da smo
 - pronašli traženi element.

Točno jedno od toga je istinito! (\implies treba ≤ 2 pitanja.)

Ako treba, potragu ponavljamo s novim l i d . Potraga traje

- sve dok nismo našli traženi element i vrijedi $l \leq d$.

U protivnom ($l > d$), nemamo više elemenata za potragu.

Binarno pretraživanje — algoritam

Algoritam:

```
nasli = 0;  l = 0;  d = n - 1;

while (!nasli && l <= d) {
    i = (l + d) / 2;
    if (elt < x[i])
        d = i - 1;
    else if (elt > x[i])
        l = i + 1;
    else
        nasli = 1;
}
```

Zadatak. Izbacite uvjet `!nasli` i iskoristite `break` gdje treba.

Binarno pretraživanje — primjer 1

Primjer. U sortiranom polju ispitajte nalazi li se broj 55.

12	18	42	44	55	67	94
----	----	----	----	----	----	----

↑
l = 0

nasli = 0

↑
d = 6

Binarno pretraživanje — primjer 1

Primjer. U sortiranom polju ispitajte nalazi li se broj 55.

12	18	42	44	55	67	94
----	----	----	----	----	----	----

↑
l = 0

↑

↑
d = 6

$$i = (l + d) / 2 = 3$$

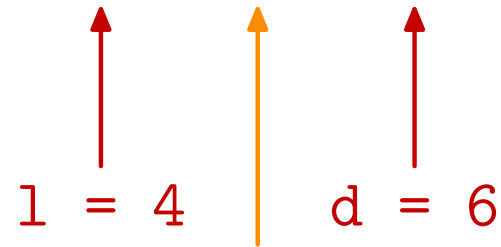
$$x[3] < 55$$

$$\text{nasli} = 0$$

Binarno pretraživanje — primjer 1

Primjer. U sortiranom polju ispitajte nalazi li se broj 55.

12	18	42	44	55	67	94
----	----	----	----	----	----	----



$$i = (1 + d) / 2 = 5$$

$$x[5] > 55$$

$$\text{nasli} = 0$$

Binarno pretraživanje — primjer 1

Primjer. U sortiranom polju ispitajte nalazi li se broj 55.

12	18	42	44	55	67	94
----	----	----	----	----	----	----



```
i = l = d = 4
```

```
x[4] == 55
```

```
nasli = 1
```

Binarno pretraživanje — primjer 2

Primjer. U sortiranom polju ispitajte nalazi li se broj 21.

12	18	42	44	55	67	94
----	----	----	----	----	----	----

↑
l = 0

nasli = 0

↑
d = 6

Binarno pretraživanje — primjer 2

Primjer. U sortiranom polju ispitajte nalazi li se broj 21.

12	18	42	44	55	67	94
----	----	----	----	----	----	----

↑
l = 0

↑

↑
d = 6

$$i = (l + d) / 2 = 3$$

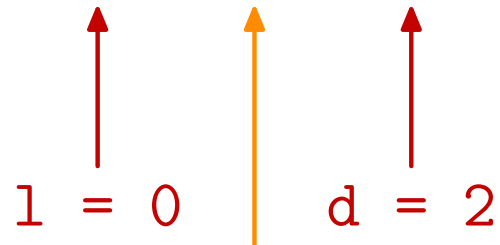
$$x[3] > 21$$

$$\text{nasli} = 0$$

Binarno pretraživanje — primjer 2

Primjer. U sortiranom polju ispitajte nalazi li se broj 21.

12	18	42	44	55	67	94
----	----	----	----	----	----	----



$$i = (l + d) / 2 = 1$$

$$x[1] < 21$$

$$\text{nasli} = 0$$

Binarno pretraživanje — primjer 2

Primjer. U sortiranom polju ispitajte nalazi li se broj 21.

12	18	42	44	55	67	94
----	----	----	----	----	----	----



```
i = l = d = 2
```

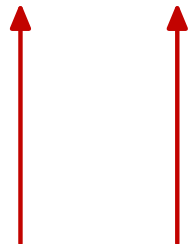
```
x[2] > 21
```

```
nasli = 0
```

Binarno pretraživanje — primjer 2

Primjer. U sortiranom polju ispitajte nalazi li se broj 21.

12	18	42	44	55	67	94
----	----	----	----	----	----	----


(d = 1) < (1 = 2)

Binarno pretraživanje — funkcija

Funkcija koja vraća odgovor (“skraćeni” oblik):

```
int binary_search(int x[], int n, int elt) {
    int l = 0, d = n - 1, i;
    while (l <= d) {
        i = (l + d) / 2;
        if (elt < x[i])
            d = i - 1;
        else if (elt > x[i])
            l = i + 1;
        else
            return 1;
    }
    return 0; }
```

Binarno pretraživanje — složenost

Koliko traje najdulja potraga (= ako element **nismo** našli)?

🔴 nakon 1. podjele — duljina niza za potragu je $\leq \frac{n}{2}$

🔴 nakon 2. podjele — duljina niza za potragu je $\leq \frac{n}{4}$

🔴 nakon k -te podjele — duljina niza za potragu je $\leq \frac{n}{2^k}$.

Zadnji prolaz k smo napravili

🔴 kad se prvi puta dogodi da je duljina pala **strogo** ispod 1, tj. u prošlom koraku je duljina niza još bila ≥ 1 . Onda je

$$\frac{n}{2^k} < 1 \quad \text{i} \quad \frac{n}{2^{k-1}} \geq 1.$$

Dakle, za zadnji prolaz k vrijedi $2^{k-1} \leq n < 2^k$.

Binarno pretraživanje — složenost (nastavak)

Složenost opet mjerimo brojem usporedbi, ali sada koristimo

● “manji (ili jednak)”, odnosno, “veći (ili jednak)”,

jer imamo uređaj među elementima i niz je sortiran po tom uređaju. Operacije na indeksima, opet, ne brojimo.

U najgorem slučaju, za broj raspolavljanja k vrijedi

$$2^{k-1} \leq n < 2^k,$$

$$k - 1 \leq \log_2 n < k,$$

ili

$$k = 1 + \lfloor \log_2 n \rfloor.$$

Svako raspolavljanje ima najviše 2 usporedbe, pa je

$$\text{broj usporedbi} \leq 2 \cdot (1 + \lfloor \log_2 n \rfloor).$$

Binarno pretraživanje — složenost (nastavak)

Zapis za trajanje:

$$T(n) \in O(\log n).$$

Značenje: trajanje, u najgorem slučaju, **logaritamski** ovisi o n .

Primjer. U sortiranom telefonskom imeniku s 10^6 osoba, dovoljno je **samo 20** raspolavljanja!

Zaključak: **Sortiramo** zato da bismo **brže** tražili!

Zadatak. Može se napraviti i varijanta sa

- 🕒 **samo jednom** usporedbom u svakom **raspolavljanju** i još **jednom** usporedbom na **kraju**.

Probajte sami (ili, v. malo kasnije)!

Zadaci za operacije s nizovima

Zadaci za operacije s nizovima

Zadaci. Napišite funkciju koja kao argument prima niz od n cijelih brojeva x_0, x_1, \dots, x_{n-1} , uz pretpostavku da je $n > 0$ (formalni argumenti su niz i njegova duljina). Funkcija treba:

- vratiti produkt svih članova niza,
- vratiti najveći član niza i njegov indeks kroz varijabilni argument,
- provjeriti postoji li član niza koji je djeljiv sa zadanim ulaznim brojem,
- provjeriti jesu li svi članovi niza jednaki zadanom ulaznom broju,
- provjeriti jesu li svi članovi niza međusobno jednaki.
- provjeriti postoje li barem dva jednaka člana niza (različitih indeksa).

Binarno pretraživanje — zadatak 1

Zadatak. Sljedeća funkcija za binarno traženje ima samo **jednu** usporedbu u svakom **raspolavljanju** i još **jednu** na **kraju**.

```
int binary_search_1_1(int x[], int n, int elt) {
    int l = 0, d = n - 1, i;
    while (l < d) {                /* Strogo manje! */
        i = (l + d) / 2;
        if (elt < x[i])
            d = i - 1;
        else                        /* elt >= x[i] */
            l = i;                  /* Ne: i + 1. */
    }
    return (elt == x[l]); }

```

Pitanje. Radi li ona **korektno** u svim slučajevima? **(NE!)**

Binarno pretraživanje — zadatak 2

Zadatak. Sljedeća funkcija je vrlo *slična*. Jedine promjene su u usporedbi `elt: < ↦ <= (+ d, 1)` i testu na kraju: `l ↦ d`.

```
int binary_search_1_d(int x[], int n, int elt) {
    int l = 0, d = n - 1, i;
    while (l < d) {                /* Strogo manje! */
        i = (l + d) / 2;
        if (elt <= x[i])
            d = i;                 /* Ne: i - 1. */
        else                       /* elt > x[i] */
            l = i + 1;
    }
    return (elt == x[d]); }       /* Moze: elt == x[l] */
```

Pitanje. Radi li ona *korektno* u svim slučajevima? (DA!)

Binarno pretraživanje — kratko objašnjenje

Upute za dokaz (ne)korektnosti:

Ako je $d \geq l + 2$, onda obje funkcije sigurno “skraćuju” niz za nastavak traženja (dio u kojem se još može nalaziti traženi element) — jer vrijedi $l < i < d$.

Ako je $l = d$ (preostao je jednočlani niz), onda obje funkcije vraćaju korektan odgovor.

Dakle, treba još provjeriti samo slučaj dvočlanog niza, kad je $d = l + 1$. Tada radimo raspolavljanje, a indeks “srednjeg” elementa je “donje cijelo”

$$i = \lfloor (l + d) / 2 \rfloor = l.$$

Zato treba biti oprezan na lijevom rubu $l = i$.

Binarno pretraživanje — objašnjenje za $i = 1$

Funkcija `1_l` “čuva” lijevi rub intervala.

- Ako je `elt < x[l]`, onda postavljamo `d = l - 1`, što prekida petlju. Konačni test `elt == x[l]` daje korektan odgovor, iako prethodni test zabranjuje(!) tu mogućnost.
- Ako je `elt >= x[l]`, onda postavljamo `l = l`. Dakle, `l` i `d` se ne mijenjaju, tj. dobivamo beskonačnu petlju!

Funkcija `1_d` “čuva” desni rub intervala. Ovdje u oba slučaja sigurno “skraćujemo” interval i prekidamo petlju.

- Ako je `elt <= x[l]`, onda postavljamo `d = l` (prekid). Konačni test `elt == x[d]` ovdje provjerava lijevi rub.
- Ako je `elt > x[l]`, onda postavljamo `l = l+1 = d` (prekid), a test `elt == x[d]` ovdje provjerava desni rub.

Dakle, uvijek vraća korektan odgovor (može i `elt == x[l]`)!