Programiranje 1

Nizovi, operacije, pretraživanje nizova

Matej Mihelčić

Prirodoslovno-matematički fakultet
Matematički odsjek

15. siječnja 2024.

Polje

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

U programskom jeziku *C*, **indeks uvijek** počinje od **nule**. Polje je slično vektoru u matematici. Npr. vektor $x = (x_1, x_2, \ldots, x_n)$ odgovara *C* polju $x = (x_0, \ldots, x_{n-1})$.

```
1 double x[3]; /* polje x s 3 clana tipa double */
2 x[0] = 0.2;
3 x[1] = 0.7;
4 x[2] = 5.5;
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];
```

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

Elementi jednodimenzionalnog polja su:
`ime[0], . . . , ime[izraz - 1]`.
Svaki **element** `ime[i]` je **varijabla** tipa **tip**.

Definicija jednodimenzionalnog polja

Deklaracija **memorijske klase nije obavezna**.

Polje deklarirano **bez memorijske klase:**

- **unutar funkcije** je **automatska** varijabla. Memorija se rezervira na **sistemskom stogu** (eng. *run time stack*) ulaskom u funkciju.
- **izvan svih funkcija** je **statička** varijabla.

Unutar neke funkcije, **polje** se može učiniti **statičkim** pomoću identifikatora memorijske klase `static` (više o tome na **Prog** 2).

Ključne informacije o polju u *C*-u

Ime polja je sinonim za **konstantni pokazivač** na **prvi element** polja (**adresa prvog elementa** polja). Više detalja o tome će biti na **Prog** 2.

Radi **efikasnosti** pristupa, **elementi polja** smještaju se, u **uzastopne memorijske lokacije**, redom prema **indeksu**.

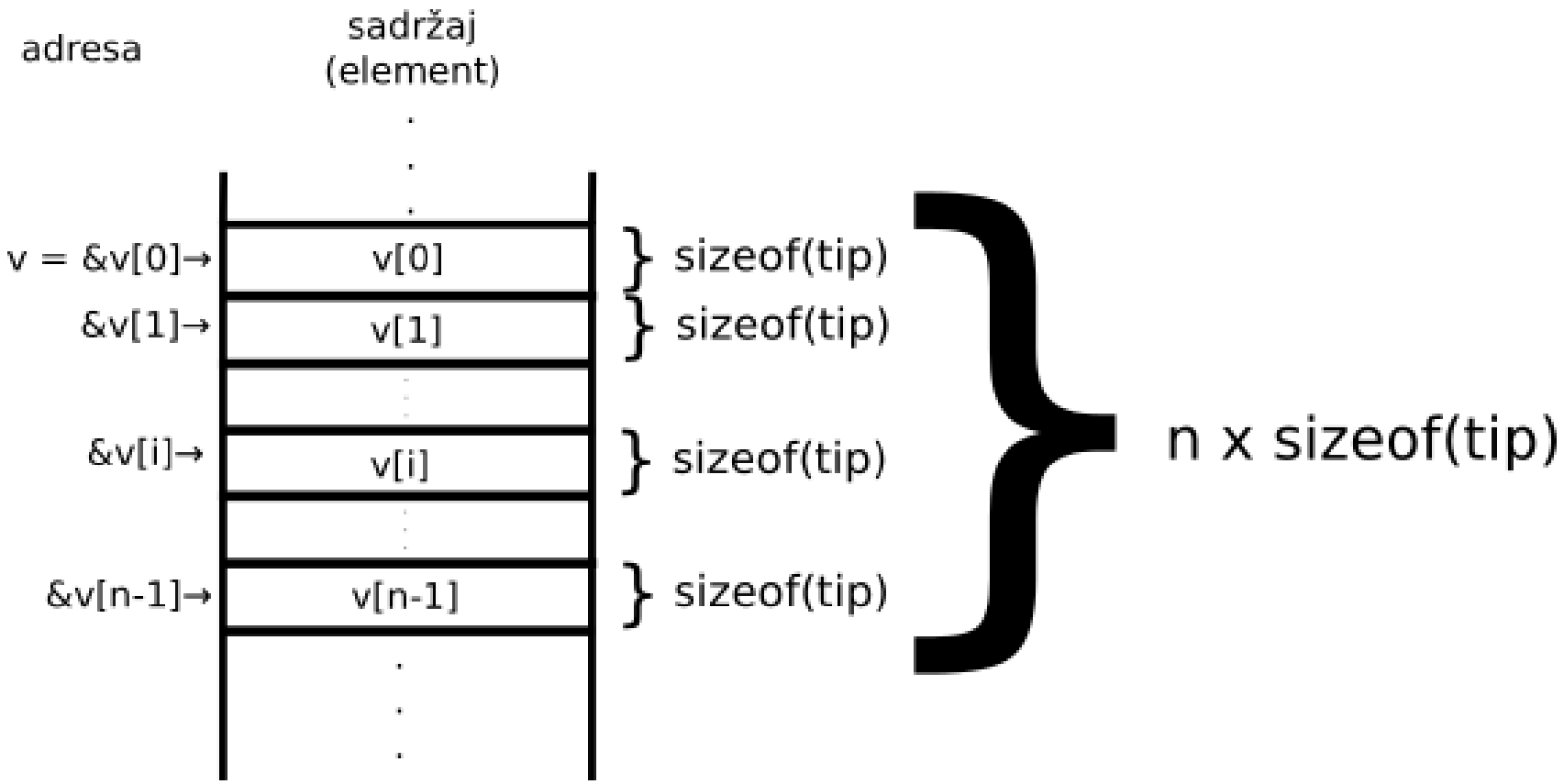
Polje u memoriji **jednoznačno** je definirano:

- **početnom adresom** polja - spremljena u varijabli **ime** polja.
- **tipom** svakog elementa - iz kojeg se može odrediti **memorijska veličina** elementa,
- **brojem** elemenata.

Adresa svakog elementa se može **izračunati** (zbog toga što su elementi polja **spremljeni u bloku** - uzastopnim memorijskim lokacijama).

Spremanje polja u memoriji i adrese elemenata

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



Adresa i-tog elementa je matematički:
$\&v[i] = v + i * sizeof(tip)$, za $i = 0, \ldots, n-1$.

Kontrola granice za indekse

U *C*-u **nema kontrole** granice za indekse!

Adresa elementa `v[i]` ovisi **samo** o:

- **početnoj adresi** polja (spremljenoj u varijabli **ime** polja),
- **tipu** elementa (**memorijskoj veličini** pojedinog elementa),
- **indeksu** elementa

Broj elemenata u polju (iz **definicije** polja) **nije bitan** za računanje adrese elementa. Broj elemenata se koristi kod **inicijalne rezervacije** memorije za polje i nigdje se **ne pamti**.

Iz tog razloga u *C*-u **nema kontrole** granica za **indeks** elementa polja. Programer **mora** voditi računa o tome da **ne pristupa** memorijskim lokacijama izvan **dozvoljenih** (rezerviranih) granica.

Indeksi se kontroliraju samo na mjestima gdje se **rezervira** memorija za polje.

Inicijalizacija polja

Polja se mogu **inicijalizirati** element po element, navođenjem popisa **vrijednosti** elemenata unutar **vitičastih** zagrada. Vrijednosti popisa **odvojene** su **zarezom** (koji **nije** operator).

Sintaksa:

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

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

```
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

Ako je **broj** inicijalizacijskih vrijednosti `n`:

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

Prilikom **inicijalizacije**, duljina polja **ne mora** biti zadana. Tada se **duljina** polja računa **automatski**, iz **broja** inicijalizacijskih vrijednosti.

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

kreira polje `v` duljine 3 elementa i **inicijalizira** ga.

Inicijalizacija polja

Polja znakova mogu se **inicijalizirati znakovnim nizovima**.

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

U gornjem primjeru definirano je polje od četiri 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:

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

Gornji dio koda je pogrešan jer **lijeva** strana **pridruživanja ne smije** biti **polje** (ime polja je **konstantni pokazivač** - sadrži adresu **prvog** elementa).

Računanje aritmetičke sredine

Primjer: treba izračunati aritmetičku sredinu elemenata polja.

```
1  #include <stdio.h>
2  int main(void) {
3       double v[] = {2.0, 3.11, 4.05, -1.07};
4       int n = sizeof(v) / sizeof(double), i;
5       double a_sredina = 0.0;
6
7       for(i = 0; i < n; ++i)
8          a_sredina += v[i];
9       a_sredina /= n;
10     printf("␣Sredina␣je␣%20.12f\n", a_sredina);
11
12     return 0;
13 }
```

Polje kao argument funkcije

Polje **može** biti **formalni** i **stvarni argument funkcije**. U tom slučaju se **ne prenosi cijelo polje po vrijednosti** (kopira polje), već funkcija **dobiva** (po vrijednosti) **pokazivač** na **neki** element polja.

Taj **pokazivač** funkcija (**lokalno**) tretira kao **pokazivač** na **prvi** element nad kojim će se izvoditi naredbe te funkcije (iako on **ne mora** sadržavati adresu **prvog** elementa tog polja).

Unutar funkcije, **elementi polja** mogu se **dohvatiti** i **promijeniti** korištenjem **indeksa** polja.
Razlog: **aritmetika pokazivača**.

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

```
1 f(tip v[], ...) ili f(tip *v, ...)
```

Polje kao argument funkcije

U **prvom** načinu **ne treba** navesti duljinu polja. **Drugi** način koristi činjenicu 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, **dodajemo** ključnu riječ `const` na početku deklaracije argumenta (npr. prvi string u `scanf`, `printf`):

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

Funkcija koja prima polje realnih brojeva i računa srednju vrijednost svih elemenata polja (elemente ne može mijenjati).

```
double sr_vrijednost(int n, const double v[]) {
   int i;
   double suma = 0.0;
   for (i = 0; i < n; ++i) suma += v[i];
return suma / n;}
```

Polje kao argument funkcije

Broj elemenata `n` je također **argument** funkcije. On je **nužan** da bi funkcija znala broj elemenata s kojima može raditi. Osim korištenjem argumenta, taj broj možemo spremiti i u globalnu varijablu.

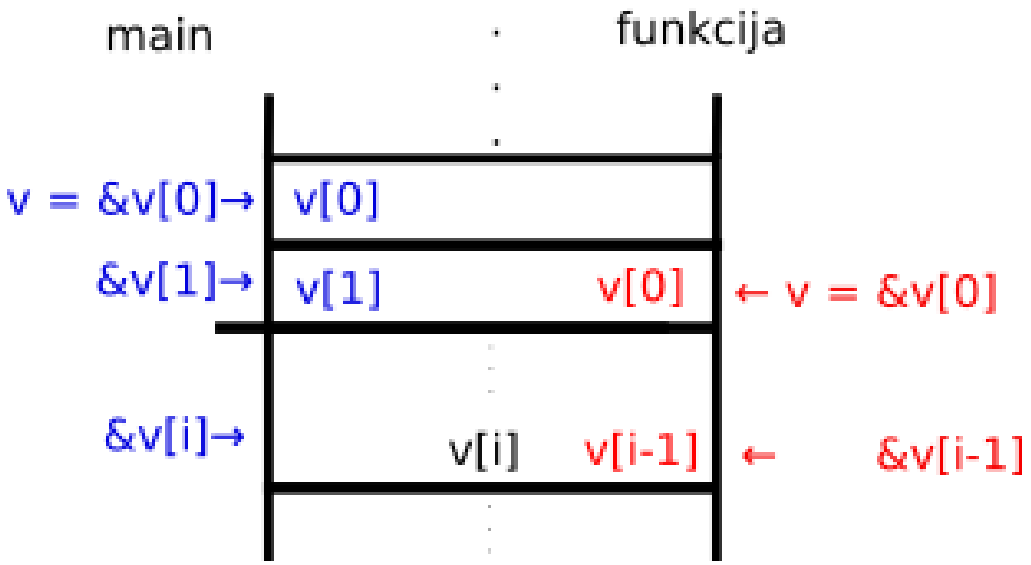
Pri **pozivu** funkcije koja ima **polje** kao **formalni** argument, **stvarni** argument je **ime polja** ili **pokazivač** na prvi element nad kojim funkcija može izvoditi operacije.

```
1 int main(void) {
2     int n;
3     double v[] = {1.0, 2.0, 3.0}, sv;
4     n = 3;
5     sv = sr_vrijednost(n, v);
6     return 0;
7 }
```

Poziv `sr_vrijednost(2, &v[1])` je **korektan**.

Polje v u glavnom programu i u funkciji

Kod poziva `srednja_vrijednost(2, &v[1])` **lokalna** varijabla `v` u **funkciji** poprima vrijednost `v = &v[1]`, za `v` iz **main**.



Aritmetika pokazivača

Ime polja je **konstantni pokazivač** na **prvi element** u polju. Za polje `niz`, `niz = &niz[0]` ili `*niz = niz[0]`.

Također, svaki **pokazivač** na neki objekt možemo interpretirati i kao **pokazivač** na **prvi element** u **polju** objekata tog tipa. Tako dobijemo vezu **pokazivač - polje** u **funkciji**.

Elementi polja spremaju se na **uzastopnim** lokacijama u memoriji. Zato, za **svaki element** polja `niz` vrijedi veza:
`niz+i = &niz[i]` ili `*(niz+i) = niz[i]`, za svaki `i`.

Stvarne adrese ovise o **memorijskoj veličini** elemenata u polju (što ovisi o **tipu** elemenata).

Pokazivači i jednodimenzionalna polja

Ime **bilo kojeg** polja (tako i **jednodimenzionalnog**) je **konstantni pokazivač** na **prvi** element polja. Zbog toga se vrijednost toga pokazivača **ne smije mijenjati**.

```
1 int a[10], b[10];
2 ...
3 a = a + 1; /* Greska, a je konst. pokazivac. */
4 b = a; /* Greska! */
```

Adresa prvog elementa se **pamti** nakon što je **alocirana** memorija za cijelo polje. Adrese se računaju koristeći **indekse** i **memorijske veličine** tipa elemenata polja.

Pokazivači i jednodimenzionalna polja

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

```
1 int a[10], *pa;
2 ...
3 pa = &a[0];
4 *(pa + 3) = 20; /* ekviv. s a[3] = 20; */
5 *(a + 1) = 10; /* ekviv. s a[1] = 10; */
```

Prioriteti i asocijativnost

Primjer: Kod korištenja pokazivača za pristupanje i promjenu elemenata polja treba pripaziti na prioritete operatora i asocijativnost.

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

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

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** memorijske lokacije na koju pokazuje `ptr`, a **ne** vrijednost samog pokazivača (adresa).

Zbog **asocijativnosti unarnih** operatora $D \rightarrow L$, isti izraz možemo zapisati i kao:

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

U gornjem primjeru se izvrši: a) **dereferenciranje**, b) **inkrementiranje**, c) korištenje **povećane** vrijednosti `*ptr`.

Važnost prioriteta i asocijativnosti

Kod **postfiks** notacije operatora **inkrementiranja**:

- za **povećavanje** ili **smanjivanje sadržaja** memorijske lokacije na koju pokazivač pokazuje, **moramo** koristiti **zagrade**.

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

- Izraz bez zagrade inkrementira adresu na koju pokazuje pokazivač `ptr` ali vraća **staru** vrijednost. Operator dereferenciranja dohvaća vrijednost memorijske lokacije na koju je `ptr` **prethodno** pokazivao.

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

Zbroj svih članova niza

Zadatak: zadan je niz (polje) od n realnih brojeva

$$x_0, x_1, \ldots, x_{n-1}$$

Treba pronaći **zbroj** svih članova niza uz pretpostavku $n > 0$.

Algoritam: uz pretpostavku da su x_i tipa `double`.

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

Algoritam radi za proizvoljan n uz dogovor `zbroj = 0` za $n \leq 0$.

Zbroj svih članova niza

Funkcija za **zbrajanje** vrijednosti elemenata niza:

```
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

Glavna funkcija programa koja sadrži poziv funkcije:

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

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

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

    return 0;
}
```

Najmanji član niza

Zadatak: treba pronaći **najmanji** član niza od n realnih brojeva

$$x_0, x_1, \ldots, x_{n-1}$$

Pretpostavka: $n > 0$ (koristi se pri inicijalizaciji).

Algoritam ispisuje vrijednost i indeks najmanjeg elementa niza.

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

for (i = 1; i < n; ++i)
    if (x[i] < min) {
        min = x[i];
        poz = i;
    }
...
printf("Najmanji clan niza: x[%d] = %f\n",
poz, min);
```

Složenost: $n - 1$ usporedbi članova niza.

Najmanji član niza

Funkcija koja vraća samo **vrijednost** najmanjeg elementa niza:

```
1  double min_clan(int n, double x[])
2  {
3       int i;
4       double min = x[0];
5
6       for (i = 1; i < n; ++i)
7           if (x[i] < min)
8               min = x[i];
9     return min;
10 }
```

DZ: Napišite funkciju koja vraća **vrijednost najmanjeg** elementa niza, te vraća **indeks** (poziciju) najmanjeg elementa kao **varijabilni** argument (ukoliko takvih ima više, vraća indeks zadnjeg u nizu).

Problem pretraživanja nizova

Problem **pretraživanja** nizova:
Za zadani niz brojeva $x_0, x_1, \ldots, x_{n-1}$ želimo **provjeriti nalazi** li se zadani element `elt` među članovima toga niza. Formalno, tražimo odgovor na pitanje: **postoji** li indeks $i \in \{0, \ldots, n-1\}$ takav da je `elt` = x_i.

Rezultat algoritma je **odgovor** na gore postavljeno pitanje, logička vrijednost: `nasli` : 1 (**istina**) ili 0 (**laž**).

Sekvencijalno pretraživanje nizova

Ako niz **nije** sortiran (nije **uređen**), koristimo **sekvencijalno pretraživanje** (ispitujemo vrijednosti svih elemenata redom).

Pretraživanje se vrši **dok** su ispunjena dva uvjeta:

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

Potraga **završava** (u najgorem slučaju - ukoliko traženi element nije u polju) nakon ispitivanja **svih** elemenata polja. Potraga može završiti i prije ukoliko se traženi element **nalazi** negdje **prije kraja** niza (npr. na **početku** niza).

Sekvencijalno pretraživanje nizova

Traženje elementa u nizu.

```
nasli = 0;
i = 0;

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

Prvi uvjet `!nasli` se može ispustiti ukoliko koristimo `break` kada pronađemo element. **Napišite** odgovarajući kod (**DZ**).

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

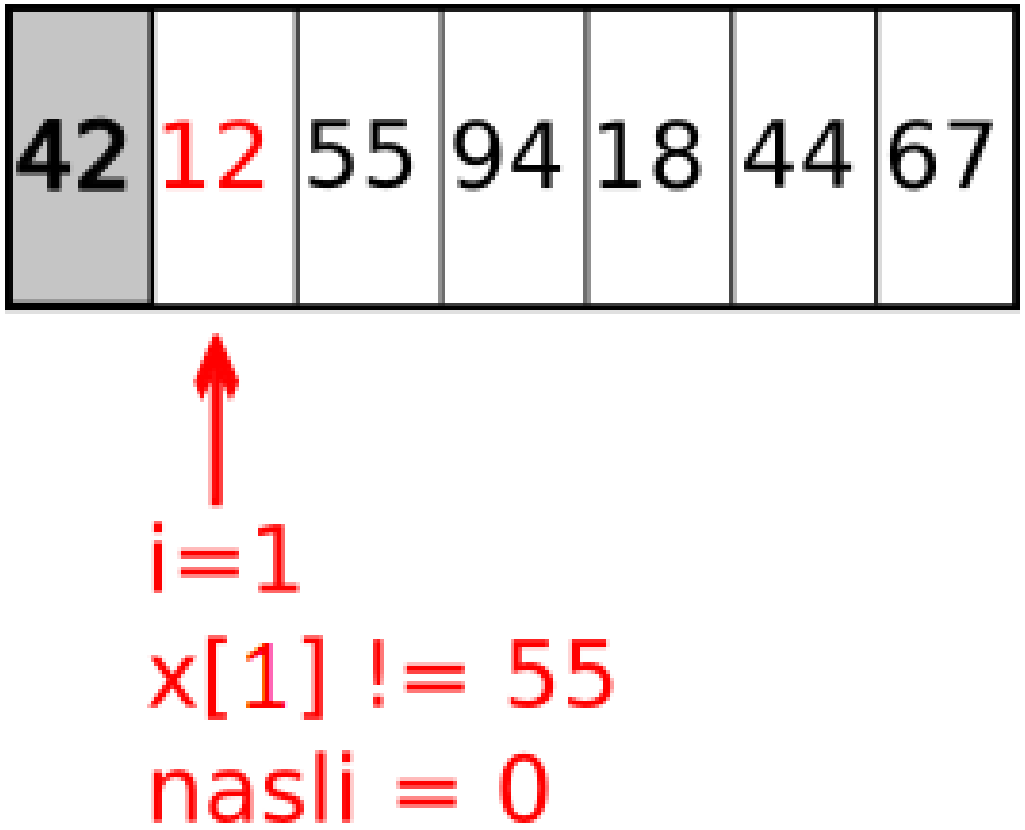
i=0

x[0] != 55

nasli = 0

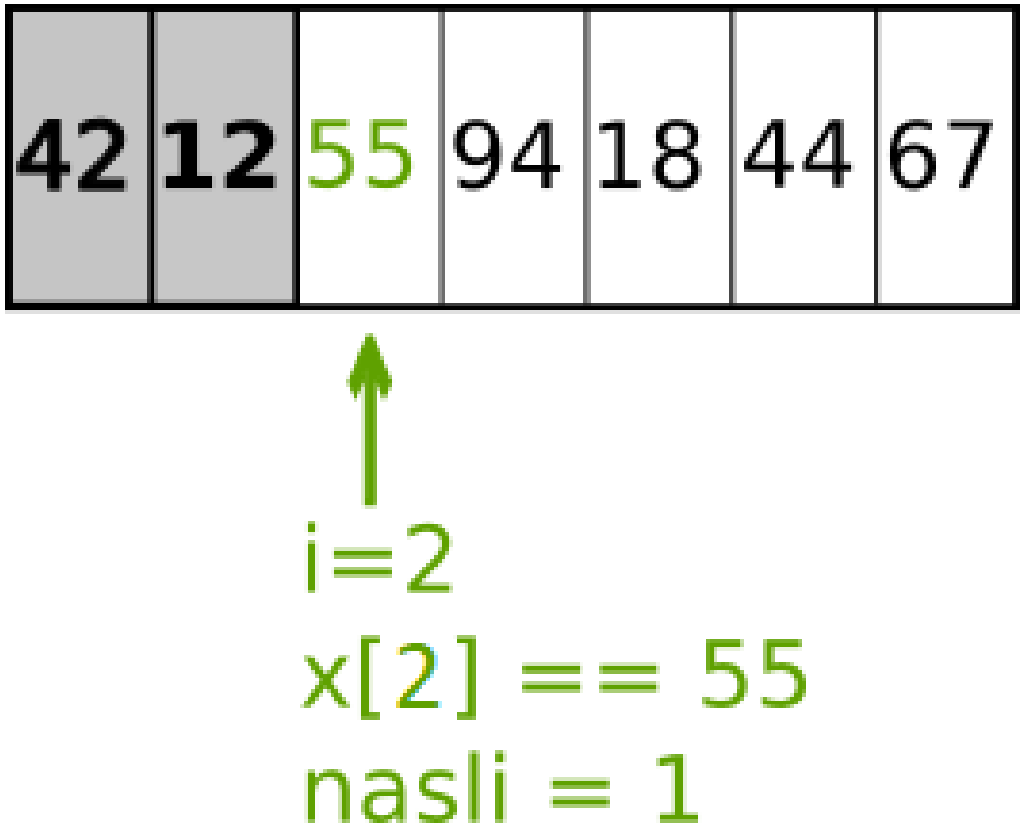
Sekvencijalno pretraživanje - primjer 1

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



Sekvencijalno pretraživanje - primjer 1

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



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

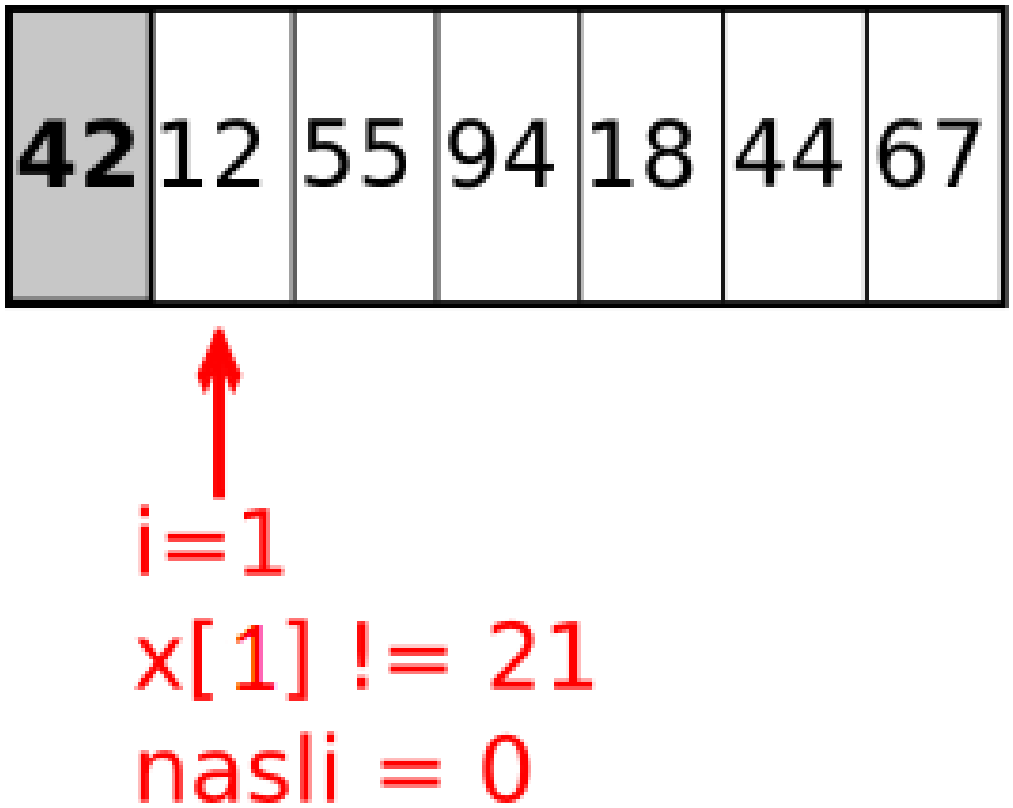
i=0

x[0] != 21

nasli = 0

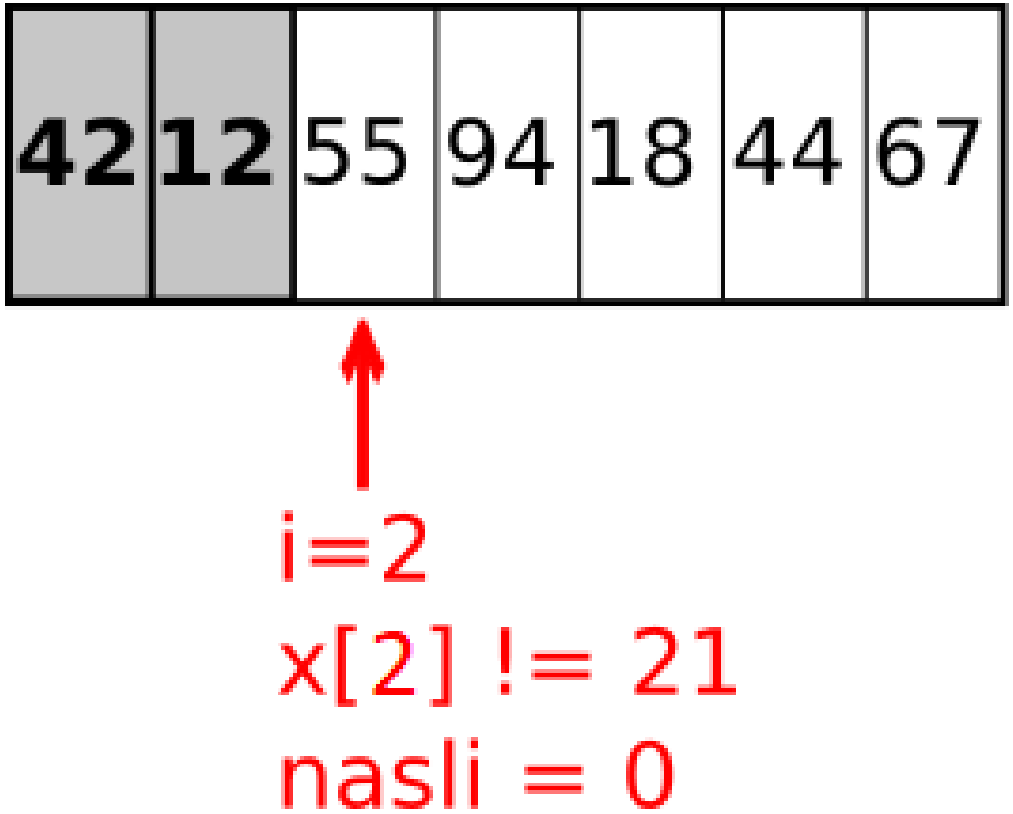
Sekvencijalno pretraživanje - primjer 2

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



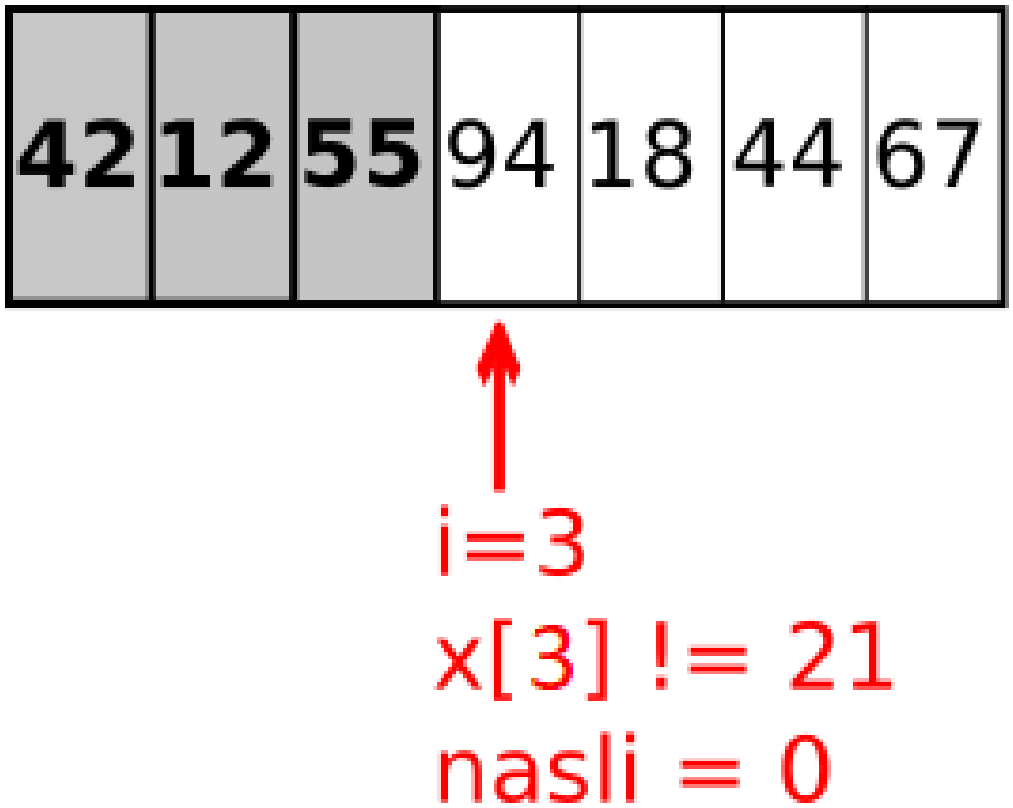
Sekvencijalno pretraživanje - primjer 2

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



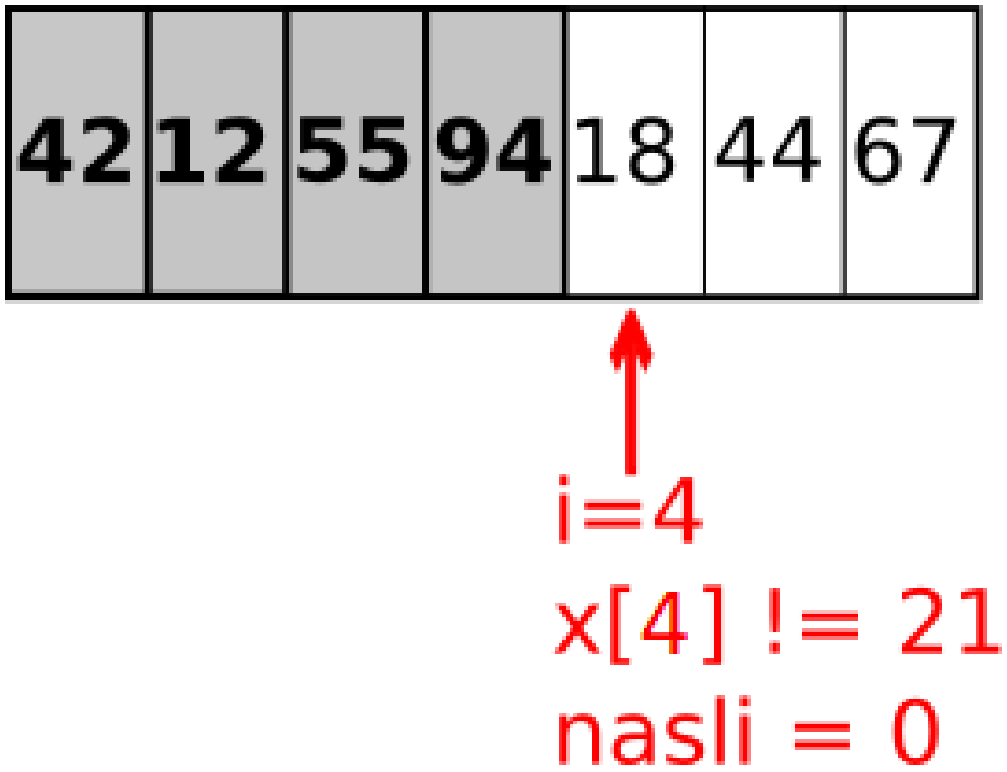
Sekvencijalno pretraživanje - primjer 2

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



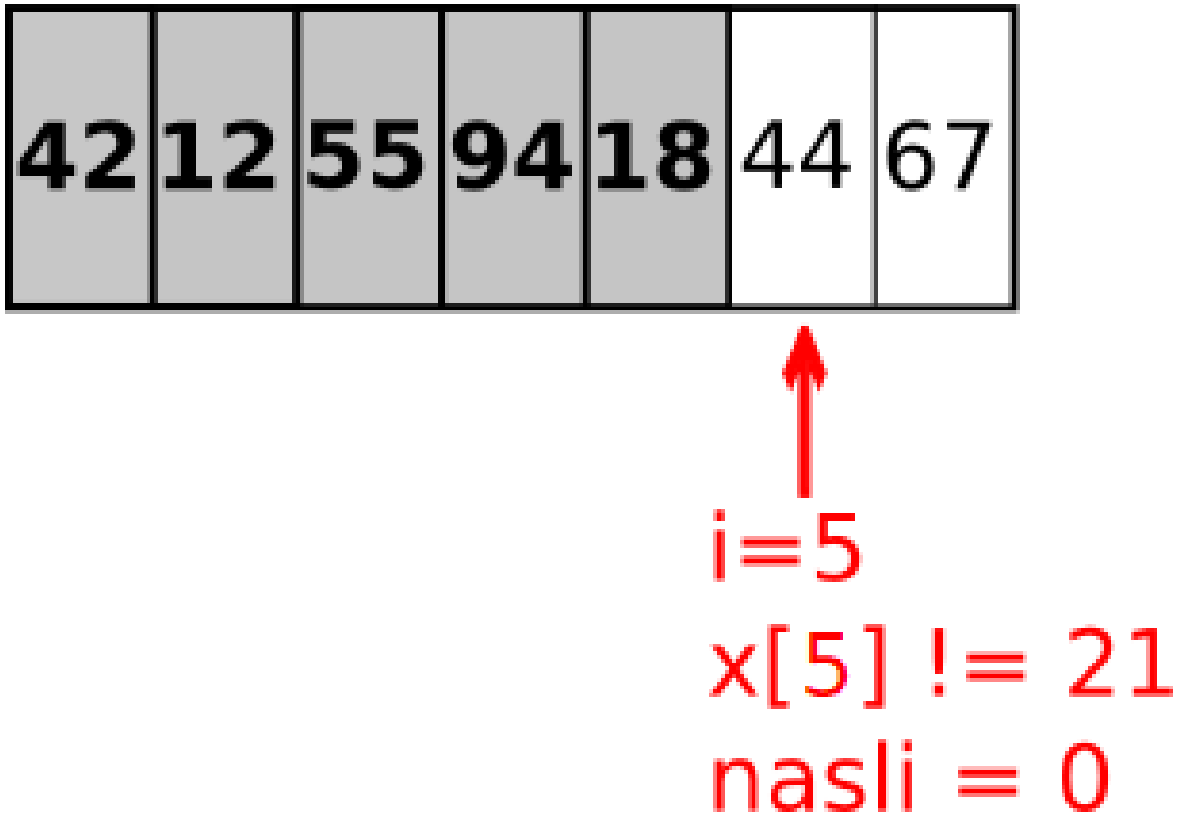
Sekvencijalno pretraživanje - primjer 2

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



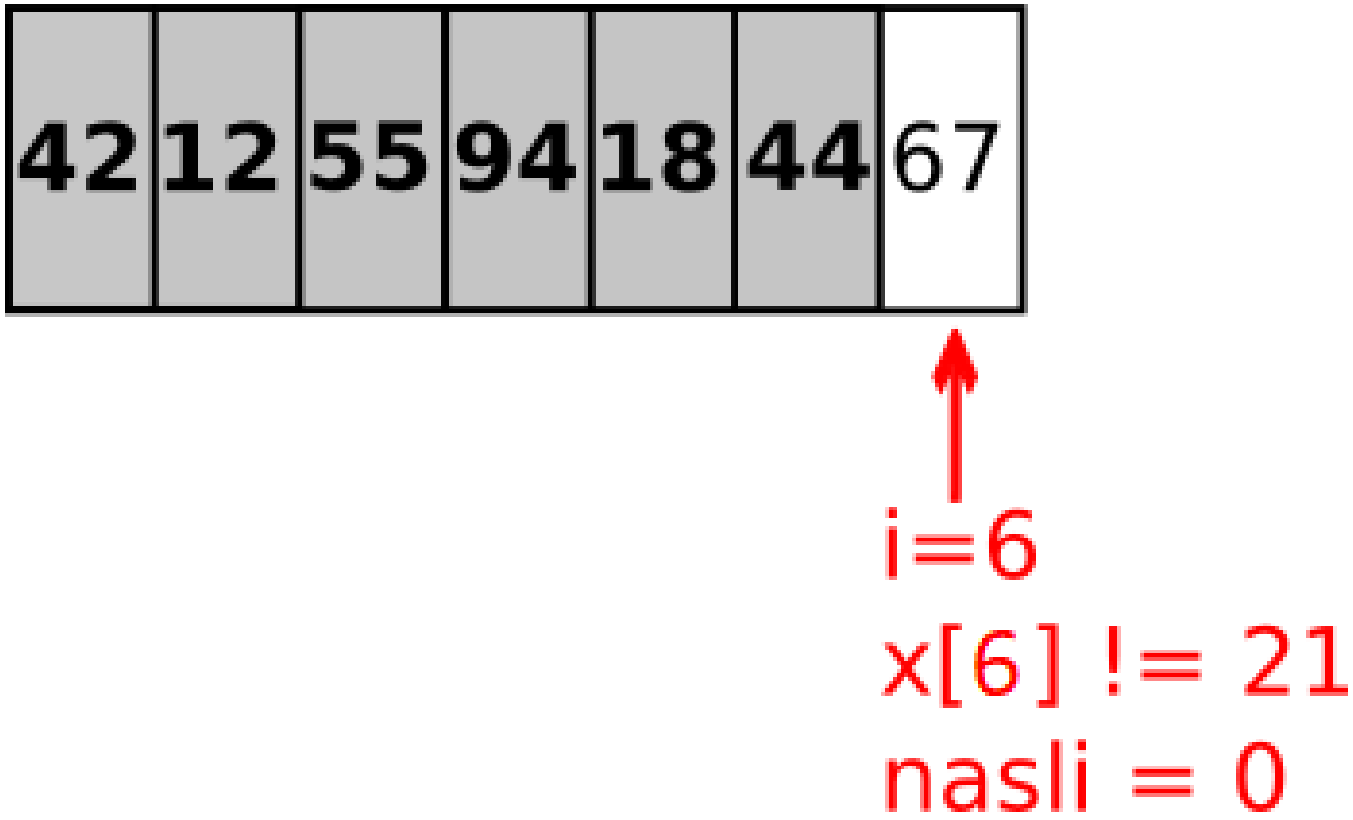
Sekvencijalno pretraživanje - primjer 2

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



Sekvencijalno pretraživanje - primjer 2

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



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 za sekvencijalno pretraživanje polja.

```
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;
}
```

Gornji kod implementira verziju algoritma koja **ne** koristi varijablu `nasli`.

Složenost sekvencijalnog pretraživanja

Složenost pretraživanja mjerimo **brojem usporedbi** *jednak* odnosno *različit* (jer **nema** uređaja), **samo** u tipu za **članove** niza. Operacije na indeksima **ne** brojimo (ima ih podjednako mnogo kao i usporedbi).

U **najgorem** slučaju moramo **provjeriti sve** članove niza, `broj_usporedbi` $\leq n$.

Ova mjera složenosti je **dobra** procjena za **trajanje** izvršavanja algoritma **sekvencijalnog** pretraživanja. Označavamo ju $T(n)$.

Trajanje zapisujemo kao: $T(n) \in \mathcal{O}(n)$.

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

Precizna definicija složenosti

Definiramo **točno** matematičko značenje zapisa:

$$T(n) \in \mathcal{O}(f(n))$$

za neke funkcije $T : \mathbb{N} \mapsto \mathbb{R}$ i $f : \mathbb{N} \mapsto \mathbb{R}$.

Ukoliko gornja tvrdnja **vrijedi**, tada **postoji** konstanta $c \in \mathbb{R}$ i **postoji** $n_0 \in \mathbb{N}$ takvi da za **svaki** $n \in \mathbb{N}$ vrijedi implikacija:

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

Značenje: T **raste sporije** od funkcije f puta neka konstanta za **sve** dovoljno velike n.

Napomena: često se piše $T(n) = \mathcal{O}(f(n))$ što **nije korektno** jer ova jednakost **nije** simetrična.

Binarno pretraživanje

Ako je niz sortiran **ulazno** ili **silazno**, tj. vrijedi: $x_0 \leq x_1 \leq \cdots \leq x_{n-1}$ ili $x_0 \geq x_1 \geq \cdots \geq x_{n-1}$, potraga se može drastično **ubrzati** tako da koristimo **binarno** pretraživanje (pretraživanje **raspolavljanjem**).

Kako bi pretraživali imenik po prezimenu da pronađemo broj određene osobe?

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

Jedno od bitnijih pitanja je **na kojem mjestu** ćemo otvoriti imenik.

Za elemente niza vrijedi:

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

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

Binarno pretraživanje

Pošto ne znamo koji su elementi u nizu, niz je najbolje podijeliti **na pola**. Time osiguramo da se `elt` podjednako vjerojatno nalazi u **prvom** ili **drugom** dijelu (prvi i drugi dio su **podjednake** veličine).

Uz gore opisani način dijeljenja niza (odabira elementa za usporedbu), bez obzira gdje se element nalazi, potragu **smanjimo** na podniz s **polovičnim** brojem elemenata.

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

$$\left\lfloor \frac{l+d}{2} \right\rfloor, \text{ ili } \left\lceil \frac{l+d}{2} \right\rceil$$

Budući da **cjelobrojnim dijeljenjem** u C-u dobijemo **prvi** izbor obično se on koristi kao **sredina**.

Binarno pretraživanje

Elemente niza `x` svrstali smo u **tri skupine**:

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

Postavljamo 3 pitanja:

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

Pošto je točno **jedna** opcija **istinita** treba nam ≤ 2 pitanja.

Ako treba potragu **ponavljamo** s **novim** l i d (tražimo dok nismo pronašli element i vrijedi $l \leq d$). U protivnom nemamo više elemenata za potragu.

Binarno pretraživanje

Algoritam binarnog pretraživanja.

```
int n = 13;
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;
}
```

DZ: Izbacite uvjet `!nasli` i iskoristite `break` gdje treba.

Binarno pretraživanje - primjer 1

Primjer: ispitajte nalazi li se broj 55 u sortiranom polju.

12	18	42	44	55	67	94

l=0 nasli = 0 d=6

Binarno pretraživanje - primjer 1

Primjer: ispitajte nalazi li se broj 55 u sortiranom polju.

12	18	42	44	55	67	94

l=0

d=6

i=(l+d) / 2 = 3

x[3] < 55

nasli = 0

Binarno pretraživanje - primjer 1

Primjer: ispitajte nalazi li se broj 55 u sortiranom polju.

12	**18**	**42**	**44**	55	67	94

l=4 d=6

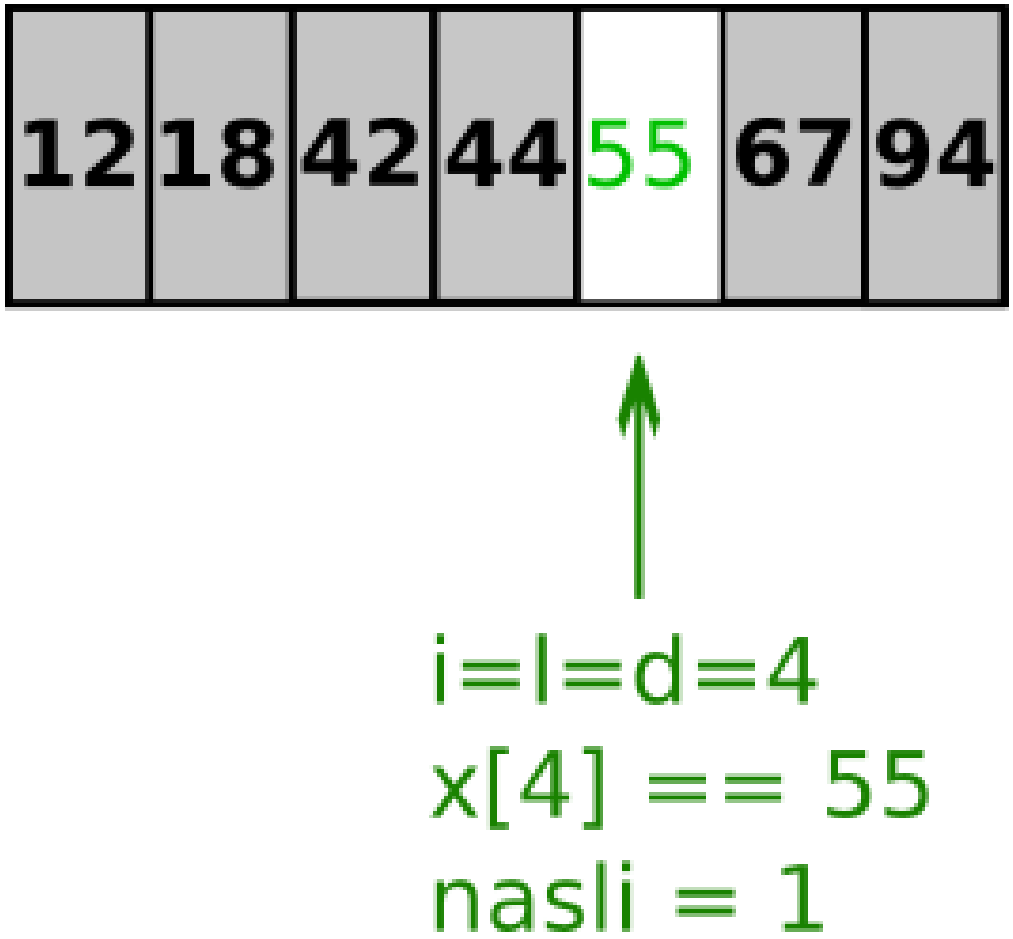
i=(l+d) / 2 = 5

x[5] > 55

nasli = 0

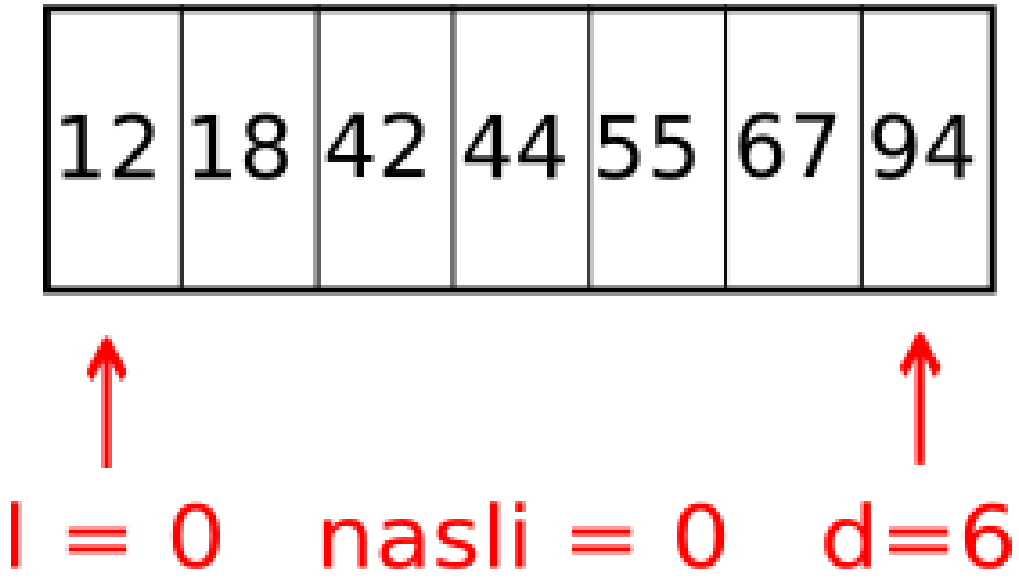
Binarno pretraživanje - primjer 1

Primjer: ispitajte nalazi li se broj 55 u sortiranom polju.



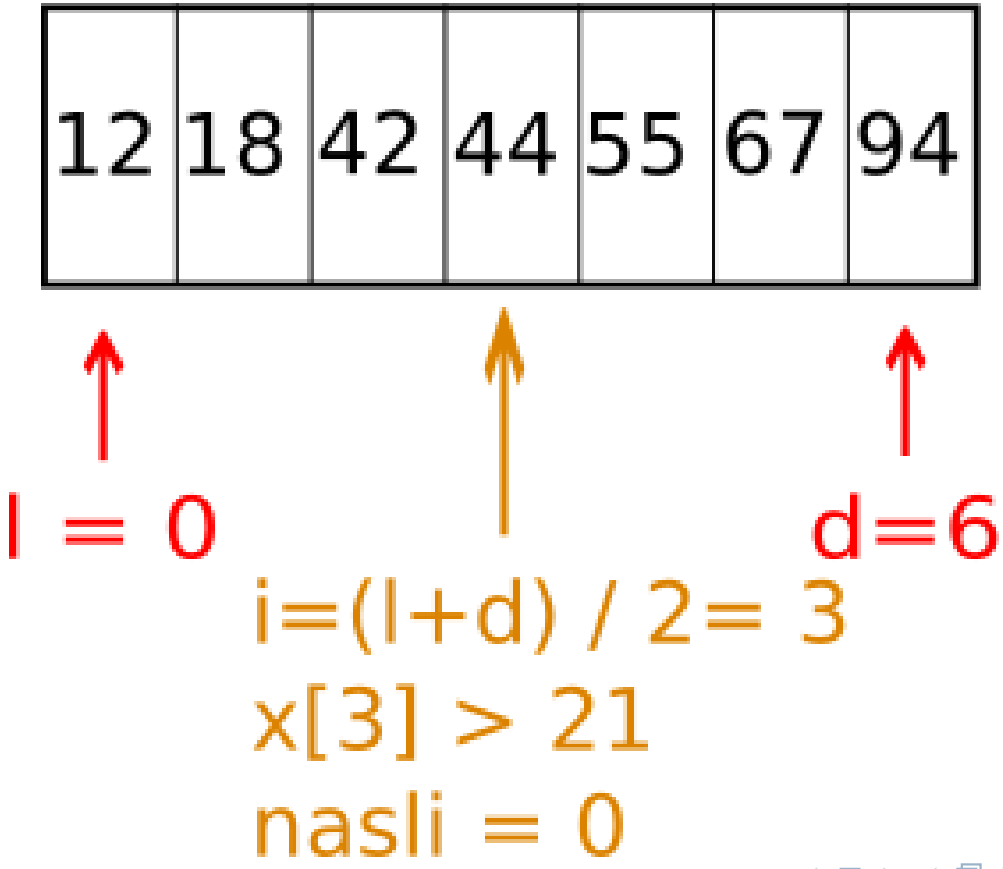
Binarno pretraživanje - primjer 2

Primjer: ispitajte nalazi li se broj 21 u sortiranom polju.



Binarno pretraživanje - primjer 2

Primjer: ispitajte nalazi li se broj 21 u sortiranom polju.



Binarno pretraživanje - primjer 2

Primjer: ispitajte nalazi li se broj 21 u sortiranom polju.

12	18	42	**44**	**55**	**67**	**94**

l = 0 d=2

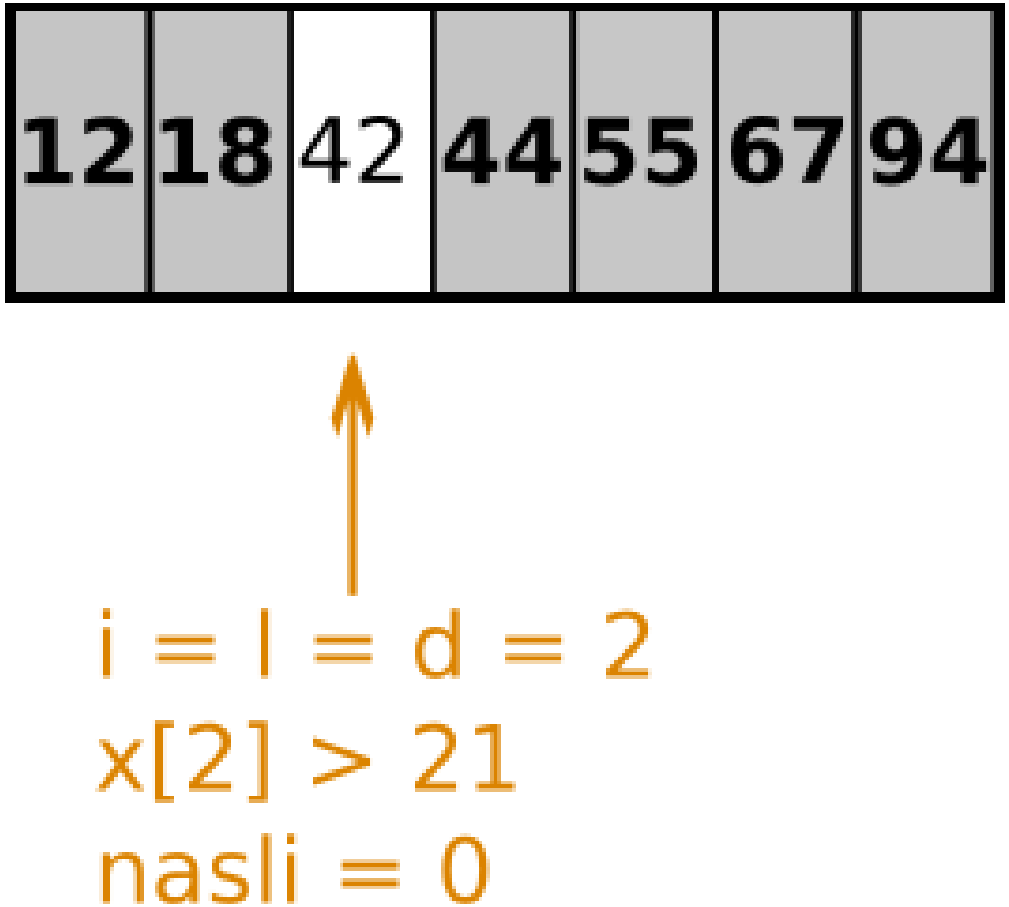
i=(l+d) / 2= 1

x[1] < 21

nasli = 0

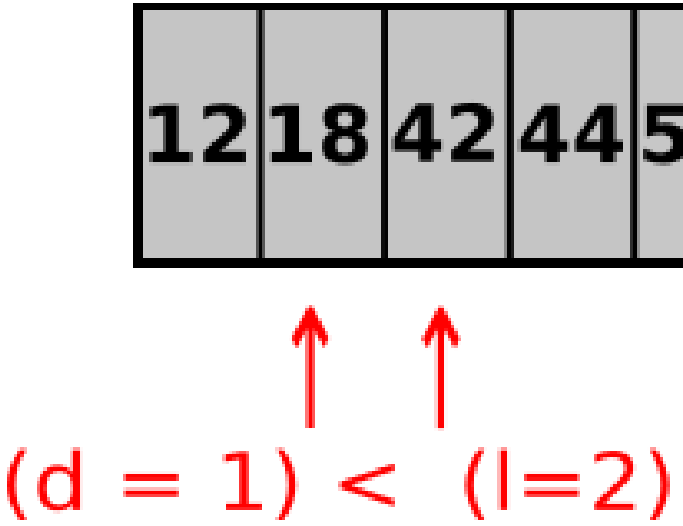
Binarno pretraživanje - primjer 2

Primjer: ispitajte nalazi li se broj 21 u sortiranom polju.



Binarno pretraživanje - primjer 2

Primjer: ispitajte nalazi li se broj 21 u sortiranom polju.



Binarno pretraživanje - funkcija

Funkcija koja vrši binarno pretraživanje polja.

```
1  int binarno_trazenje(int x[], int n, int elt) {
2
3  int l = 0, d = n - 1, i;
4
5  while (l <= d) {
6        i = (l + d) / 2;
7
8        if (elt < x[i])
9           d = i - 1;
10       else if (elt > x[i])
11          l = i + 1;
12       else
13          return 1;
14 }
15 return 0; }
```

Složenost binarnog pretraživanja

Zanima nas **koliko** traje **najdulja** potraga (ako element **nismo** pronašli).

- nakon 1. podjele, duljina niza za pretraživanje je $\leq \frac{n}{2}$,
- nakon 2. podjele, duljina niza za pretraživanje je $\leq \frac{n}{4}$,
- nakon k. podjele, duljina niza za pretraživanje je $\leq \frac{n}{2^k}$.

Zadnji prolaz k smo napravili kada se **prvi** puta dogodi da je duljina pala **strogo** ispod 1 (u prethodnom koraku je još bila ≥ 1). Vrijedi:

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

Za **zadnji** prolaz k vrijedi $2^{k-1} \leq n < 2^k$.

Složenost opet mjerimo **brojem usporedbi**, ali koristimo **manji** (**ili jednak**) i **veći** (**ili jednak**) jer **postoji uređaj** među elementima i niz je **sortiran** po tom uređaju. Operacije na indeksima **ne brojimo**.

Složenost binarnog pretraživanja

U **najgorem** slučaju, za broj **raspolavljanja** k vrijedi:

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

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

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

Svako raspolavljanje ima **najviše** 2 usporedbe, stoga vrijedi: `broj_usporedbi` $\leq 2 \cdot (1 + \lfloor log_2 n \rfloor)$.

Vrijedi: $T(n) \in \mathcal{O}(logn)$ (trajanje u najgorem slučaju **logaritamski** ovisi o n).

Primjer: u sortiranom telefonskom imeniku s 10^6 osoba dovoljno je **samo** 20 raspolavljanja.

Zaključak: **sortiramo** da bismo **brže tražili**.

Zadaci za operacije s nizovima

Zadatak: Implementirajte varijantu algoritma za binarno traženje koja radi **samo jednu** usporedbu u svakom **raspolavljanju** i **jednu** usporedbu na kraju.

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

- **vratiti umnožak** 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).