Programiranje 1
12. predavanje

Saša Singer

singer@math.hr

web.math.hr/~singer

PMF – Matematički odjel, 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

Termini "bitnih" događaja (valjda službeno):

- 2. kolokvij — utorak, 19. 1. 2010., u 12 sati,
- Popravni kolokvij — utorak, 2. 2. 2010., u 12 sati.

Upisi ocjena i usmeni (po želji):

- pogledati obavijest na rezultatima kolokvija!

Lijepo molim:

- doći u nekom od navedenih termina, bit će ih dovoljno!

Ako ne možete (bolesni i sl.) — javite se (mailom).

- Nemojte dolaziti izvan termina, i
- nemojte "zaboraviti" na ocjenu.

Naknadni upis ocjene = naknadni potpis = molba, a to košta!

Informacije — nastavak

Bitno: Aplikacija za “zadaće” se

- zaključava s početkom drugog kolokvija.

Nakon toga,

- nema više novih “računa” (prijava),
- ni predaje zadataka.

U tom trenu vrijedi:

- Tko je “unutra” i koliko je predao/la . . . , to je to,

i nema iznimaka!

Informacije — Praktični kolokvij

Praktični kolokvij (prvi krug) — kratki komentar:

- Rezultati su "jadni" — oko 61% svih prijavljenih za PK,
- tj. prošlo vas je 140 od 228.

Oko 100 studenata ima pravo na popravak.

Napomena: Nedolazak na bilo koji PK treba opravdati ispričnicom (nastavniku),

- inače gubite pravo na popravak!

Praktični kolokvij (drugi krug) — termini su:

- subota, 5. 12. i ponedjeljak, 7. 12., u praktikumu 2.

Praktični kolokvij (treći krug) ide odmah iza — termini su:

- petak, 11. 12. i subota, 12. 12., u praktikumu 2.

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 niz varijabli istog tipa (sa zajedničkim imenom) numeriranih cjelobrojnim indeksom.

- Indeks uvijek počinje od nule.
- Radi efikasnosti pristupa, elementi polja smještaju se u uzastopne memorijske lokacije (redom po indeksu).

Primjer.

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

Definicija polja

Jednodimenzionalno polje definira se na sljedeći način:

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

gdje je:

- `mem_klasa` memorijska klasa cijelog polja,
- `tip` tip podatka svakog elementa polja,
- `ime` ime polja (zajednički dio imena svih elemenata),
- a `izraz` konstantan, cjelobrojni, pozitivan izraz koji zadaje broj elemenata.

Ovaj `izraz` je najčešće pozitivna konstanta ili simbolička konstanta.

Definicija polja (nastavak)

Elementi jednodimenzionalnog polja su:

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

Svaki element je 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 funkcije polje se može učiniti statičkim pomoću identifikatora memorijske klase `static`.

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

Prilikom inicijalizacije dimenzija polja ne mora biti zadana.

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

Primjer. Možemo pisati

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

što kreira polje `v` dimenzije 3 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 pointer — adresa prvog elementa).

Primjer — aritmetička sredina

Primjer. Računanje aritmetičke sredine.

```
int main(void) {
    int i, n;
    double a_sredina = 0.0;
    double v[] = {2.0, 3.11, 4.05, -1.07};
    n = sizeof(v) / 8;

    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

Zapamtiti: Ime polja je sinonim za

- konstantni pokazivač koji sadrži adresu prvog elementa polja (više u drugom semestru).

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 prvi element polja.

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 uzima polje `v` tipa `tip` kao argument, možemo deklarirati na dva načina:

```
f(tip v[])     ili     f(tip *v)
```

U prvom načinu ne treba navesti dimenziju. 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:

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

Polje kao argument funkcije (nastavak)

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

```
double srednja_vrijednost(int n, 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 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" 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!

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 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! */
```

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

Primjer. Važnost prioriteta — ako je

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

onda je

izraz	x	ptr
`x = *ptr;`	0	1245040
`x = *ptr++;`	0	1245044
`x = (*ptr)++;`	10	1245044
`x = *++ptr;`	20	1245048
`x = ++(*ptr);`	21	1245048

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

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

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 clanova 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;
    double v[] = {1.2, 2.6, 1.8. 4.4, 0.8};

    printf("Zbroj svih clanova niza = %f.\n",
            zbroj_clanova(5, v) );

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

    return 0;
}
```

Najmanji član niza

Tražimo najmanji član niza od n realnih brojeva

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

Pretpostavka je opet $n > 0$. Ovdje se "tvrdo" koristi za korektnu inicijalizaciju.

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 clan niza: x[%d] = %f.\n",
        poz, min);
```

Složenost: $n - 1$ usporedbi.

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 u općem obliku:

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

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

Drugim riječima, pitanje glasi:

- postoji li neki indeks i takav da je `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 — od 0 do $n - 1$.

Očito, potraga je završena (u najgorem slučaju)

- nakon točno jednog prolaza kroz sve elemente.

Ona može završiti i prije, ako se traženi element nalazio negdje na početku niza.

Sekvencijalno pretraživanje — algoritam

Algoritam:

```
nasli = 0;
i = 0;

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

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 ispitajte 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 mjerimo brojem usporedbi

- "jednak", odnosno, "različit".

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.

Zapis:

$$T(n) = \mathcal{O}(n).$$

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

Točno značenje zapisa složenosti

Točno matematičko značenje zapisa

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

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

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.

Binarno pretraživanje

Ako je niz uzlazno ili silazno sortiran,

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

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 naše elemente 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 koji ćemo usporediti s elementom `elt`.

Kako ne znamo koji su elementi u nizu,

- niz je najbolje podijeliti "na pola",

jer je podjednako vjerojatno da se `elt` nalazi u prvom ili drugom dijelu.

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:

- `elt` $< x_i$? Odgovor "da" znači da nastavljamo tražiti
  - u podnizu s indeksima od l do $d = i - 1$,
- `elt` $> x_i$? Odgovor "da" znači da nastavljamo tražiti
  - u podnizu s indeksima od $l = i + 1$ do d,
- `elt` $= x_i$? Odgovor "da" znači da smo
  - pronašli traženi element.

Točno jedno od toga je istinito!

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

- nismo našli traženi element i vrijedi $l \leq d$.
- U protivnom, nemamo više elemenata za potragu.

Binarno pretraživanje — algoritam

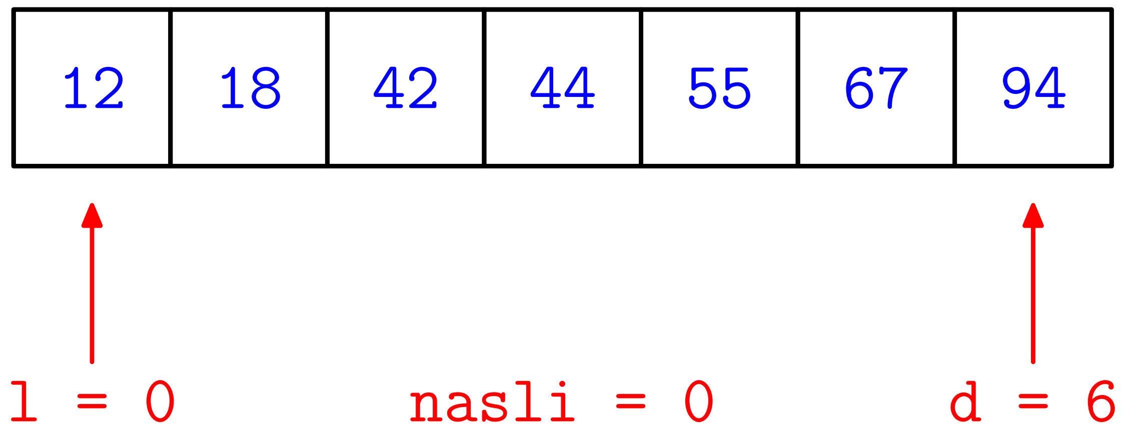
Algoritam:

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

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;
}
```

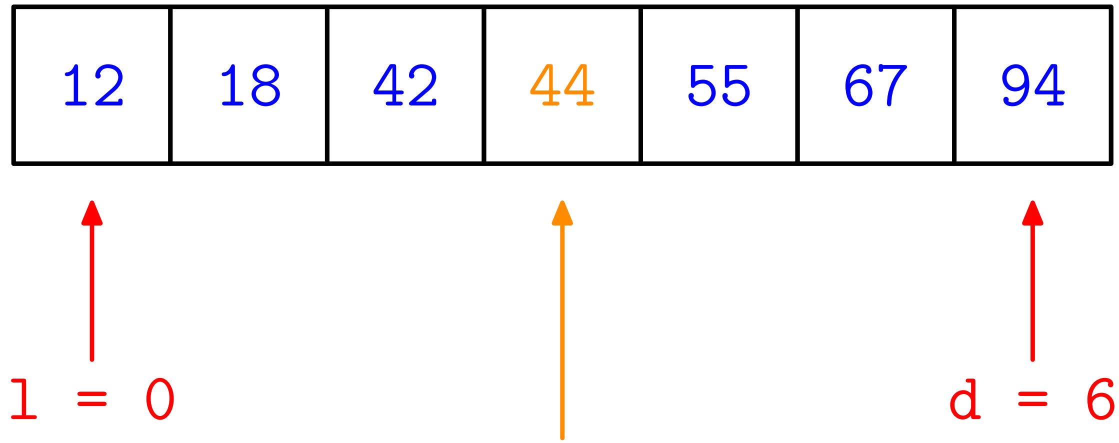
Binarno pretraživanje — primjer 1

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



Binarno pretraživanje — primjer 1

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




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

x[3] < 55

nasli = 0
```

Binarno pretraživanje — primjer 1

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

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. 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

nasli = 0
```

Binarno pretraživanje — primjer 2

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

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. 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) < (l = 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 $\leq \dfrac{n}{2}$
- nakon 2. podjele — duljina niza za potragu je $\leq \dfrac{n}{4}$
- nakon k-te podjele — duljina niza za potragu je $\leq \dfrac{n}{2^k}$.

Zadnji smo prolaz napravili kad je

$$\frac{n}{2^k} < 1,$$

dakle, $n < 2^k$, odnosno, $k > \log_2 n$. Pritom, stajemo za najmanji takav k — tj., kad je $2^{k-1} \leq n < 2^k$.

Binarno pretraživanje — složenost (nastavak)

Složenost opet mjerimo brojem usporedbi, ali sada koristimo

- "manji (ili jednak)", odnosno, "veći (ili jednak)".

U najgorem slučaju, za broj raspolavljanja k vrijedi

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

ili

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

Svako raspolavljanje ima najviše 2 usporedbe, pa je

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

Binarno pretraživanje — složenost (nastavak)

Može se napraviti i varijanta sa samo jednom usporedbom u svakom raspolavljanju (probajte sami).

Zapis za trajanje:

$$T(n) = \mathcal{O}(\log n).$$

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

Zaključak: Sortiramo zato da bismo brže tražili!