Algoritmi i strukture podataka

- predavanja -

3. Algoritmi i složenost

A priori i *a posteriori* analiza algoritma

- analiza *a priori* (teorijska)
  - trajanje izvođenja algoritma (*u najlošijem slučaju*) kao vrijednost funkcije relevantnih argumenata (npr. *broja podataka*)
- analiza *a posteriori* (empirijska)
  - statistika dobivena mjerenjem na računalu
- izbor skupova podataka za iscrpno testiranje algoritma:
  - ponašanje u najboljem slučaju (engl. *best-case scenario*)
  - ponašanje u najlošijem slučaju (engl. *worst-case scenario*)
  - prosječno (tipično) ponašanje (engl. *average-case scenario*) – za slučajan skup podataka

Analiza složenosti algoritama (analiza *a priori*) (1)

- **svrha**
  - predviđanje vremena izvođenja
  - pronalaženje učinkovitijih algoritama
  - intelektualna razonoda?

Analiza složenosti algoritama (analiza *a priori*) (2)

- pretpostavke za analizu složenosti: **slijedno jednoprocesorsko računalo (RAM model)**
  - fiksno vrijeme dohvata sadržaja memorijske lokacije
  - vrijeme obavljanja operacija (aritmetičke, logičke, pridruživanje, poziv funkcije, dohvat člana polja) je ograničeno nekom konstantom kao gornjom granicom
  - pretpostavka da su riječi u memoriji računala konstantne duljine (npr. 32 bita)
    - ako bi cijeli ili realni brojevi bili prikazani varijabilnom duljinom riječi u memoriji, onda se aritmetičke operacije ne bi obavljale u konstantnom vremenu

Analiza složenosti algoritama (analiza *a priori*) (3)

- vrijeme izvođenja ***T*** (engl. *running time*)
  - ukupno vrijeme potrebno za izvođenje svih naredbi koje treba obaviti
    (tj. broj koraka koje treba obaviti)
  - definirati neovisno o računalnoj arhitekturi, programskom jeziku, prevoditelju
  - neka je konstantno vrijeme c_i izvođenja potrebno za izvođenje *i*-te jednostavne operacije
    - općenito vrijedi $c_i \neq c_j$ za operacije *i* i *j*, ali možemo uvesti *c* tako da je $c \geq c_i$ za svaki c_i

Analiza složenosti algoritama (analiza *a priori*) (4)

- veličina ulaznog skupa podataka (engl. *input size*)
  - definira se ovisno o tipu ulaznih podataka
  - primjeri
    - ako sortiramo polje od *n* članova, onda je *n* veličina ulaznog skupa podataka, a vrijeme izvođenja označavamo ***T*(*n*)**
    - kod matrica je veličina ulaznog skupa podataka opisana s dva parametra: brojem redaka (*m*) i brojem stupaca (*n*), pa vrijeme izvođenja može biti funkcija dva parametra ***T*(*m*, *n*)**

Primjer određivanja vremena izvođenja (1)

```
a) x += y;
```

$T = c_1$

c_1 – vrijeme potrebno za obavljanje *x += y*

```
b) for (i = 1; i <= n; i++) {
       x += y;
   }
```

$T = T(n)$

$= c_1 + c_2(n + 1) + c_3 n + c_4 n$

$= (c_2 + c_3 + c_4)n + (c_1 + c_2)$

$= an + b$

c_1 – vrijeme potrebno za pridruživanje *i = 1*

c_2 – vrijeme potrebno za usporedbu *i <= n*

c_3 – vrijeme potrebno za operaciju *x += y*

c_4 – vrijeme potrebno za povećanje vrijednosti *i* za 1 (operacija *i++*)

n – broj ponavljanja naredbi *i++* i *x += y*

n +1 – broj ponavljanja usporedbe *i <= n*

T(*n*) je linearna funkcija varijable *n*.

Primjer određivanja vremena izvođenja (2)

```
c) for (i = 1; i <= n; i++) {
     for (j = 1; j <= n; j++) {
       x += y;
     }
   }
```

c_1 – vrijeme potrebno za pridruživanje $i = 1$
c_2 – vrijeme potrebno za usporedbu $i <= n$
c_3 – vrijeme potrebno za operaciju i++
c_4 – vrijeme potrebno za pridruživanje $j = 1$
c_5 – vrijeme potrebno za usporedbu $j <= n$
c_6 – vrijeme potrebno za operaciju j++
c_7 – vrijeme potrebno za operaciju $x += y$
n – broj ponavljanja operacije i++
n+1 – broj ponavljanja usporedbe $i <= n$
t_i – broj ispitivanja uvjeta unutarnje for-petlje u i-toj iteraciji vanjske for-petlje; $t_i = n+1$

$$
\begin{aligned}
T &= T(n) = \\
&c_1 + c_2(n+1) + c_3 n \\
&+ c_4 \sum_{i=1}^{n} 1 + c_5 \sum_{i=1}^{n} t_i \\
&+ c_6 \sum_{i=1}^{n} (t_i - 1) \\
&+ c_7 \sum_{i=1}^{n} (t_i - 1) \\
&= c_1 + c_2(n+1) + c_3 n \\
&+ c_4 n + c_5 n(n+1) \\
&+ c_6 n(n+1-1) \\
&+ c_7 n(n+1-1) \\
&= (c_1 + c_2) \\
&+ (c_2 + c_3 + c_4 + c_5) n \\
&+ (c_5 + c_6 + c_7) n^2 \\
&= an^2 + bn + c
\end{aligned}
$$

$T(n)$ je kvadratna funkcija varijable n.

Primjer određivanja vremena izvođenja (3)

- promatramo kako se ponaša vrijeme izvođenja kad broj ulaznih podataka postane dovoljno velik
- primjer: računanje inverzne matrice od *n* x *n* elemenata „školskim” načinom
  - vrijeme izvođenja kao funkcija od *n*: vrijeme izvođenja raste s n^3
  - npr. ako računanje inverzne matrice 100 x 100 elemenata traje 1 minutu,

    onda će računanje inverzne matrice 200 x 200 elemenata

    trajati 8 minuta ($2^3 = 8$)

Određivanje vremena izvođenja – slijedno pretraživanje (1)

Searching.cpp

- slijedno (linearno) pretraživanje polja *A* od *n* članova
  - podatci u polju ne moraju biti sortirani (tj. poredani uzlazno ili silazno)
  - naziva se još i linearno (serijsko, sekvencijalno) pretraživanje

```
za i = 0 do n-1
ako je član polja A[i] jednak traženom broju
     traženi član je pronađen
      iskoči iz petlje
```

Određivanje vremena izvođenja – slijedno pretraživanje (2)

- slijedno (linearno) pretraživanje polja *A* od *n* članova
  - podatci u polju ne moraju biti sortirani (tj. poredani uzlazno ili silazno)

```
for (i = 0; i < n; i++) {
  if (trazeni == A[i]) {
    break;
  }
}
```

Najbolji slučaj:

Traženi broj se nalazi na početku polja (član A[0]).

Najlošiji slučaj:

Traženi broj se ne nalazi u polju A.

Određivanje vremena izvođenja – slijedno pretraživanje (3)

- slijedno (linearno) pretraživanje polja *A* od *n* članova
  - podatci u polju ne moraju biti sortirani

```
for (i = 0; i < n; i++) {
  if (trazeni == A[i]) {
    break;
  }
}
```

c_1 – vrijeme potrebno za pridruživanje $i = 0$

c_2 – vrijeme potrebno za usporedbu $i < n$

c_3 – vrijeme potrebno za usporedbu *trazeni == A[i]*

c_4 – vrijeme potrebno za povećanje vrijednosti *i* za 1 (operacija *i++*)

c_5 – vrijeme potrebno za skok iz petlje (*break*)

Najbolji slučaj:

Traženi broj se nalazi na početku polja (član A[0]).

$\boldsymbol{T(n)} = c_1 \cdot 1 + c_2 \cdot 1 + c_3 \cdot 1 + c_5 \cdot 1 = c$

Najlošiji slučaj:

Traženi broj se ne nalazi u polju A.

$\boldsymbol{T(n)} = c_1 \cdot 1 + c_2 \cdot (n + 1) + c_3 \cdot n + c_4 \cdot n$

$= (c_1 + c_2) + (c_2 + c_3 + c_4) \cdot n$

$= an + b$

Slijedno pretraživanje korištenjem *strategije* - objektno oblikovnog obrasca

Searching.cpp

- **obrazac Strategija – ideja**
  - koristi se kada postoji obitelj algoritama koji se mogu primjenjivati nad nekim skupom podataka u sličnom kontekstu, a možemo ih predstaviti nekim zajedničkim konceptom

Primjer:

- pretraživanje polja na različite načine (slijedno, binarno, …)
- svaki od načina pretraživanja je jedna konkretna strategija predstavljena odgovarajućim razredom (npr. `LinearSearch`, `BinarySearch`, …)
- konkretne strategije se ponašaju slično → nasljeđuju osnovni razred `ISearch`

Slijedno pretraživanje korištenjem *strategije* – implementacija (1)

Searching.cpp

- osnovni razred: **`ISearch`** - ima ulogu sučelja (engl. *interface*)
- sučelje je implementirano kao apstraktni razred (sadrži barem jedan čisti virtualni funkcijski član):

```
virtual RetValSearch search(const T A[], size_t n,
                            const T& item) = 0;
```

- virtualni funkcijski član `search` treba biti definiran u razredima koji ga nasljeđuju
- razredi koji ga nasljeđuju predstavljaju različite taktike pretraživanja polja: **`LinearSearch`**, **`BinarySearch`**, **`JumpSearch`**, itd.
- za sučelje se ne može instancirati objekt

Slijedno pretraživanje korištenjem *strategije* – implementacija (2)

Searching.cpp

- Kako postaviti strategiju?
- razred koji postavlja i poziva neku od strategija: **`ContextSearch`**
  - sadrži privatni podatkovni član: **`ISearch<T>* searchStrategy`**
  - `searchStrategy` - pokazivač na osnovni razred `ISearch`, preko kojeg se poziva odabrana funkcija `search` iz nekog od razreda
- glavni program u kojemu se odabire neka od strategija pretraživanja i poziva pretraživanje za zadano polje:

```
ContextSearch<int> cs;
cs.setSearch(type); // linearno, blokovsko, …
// traži vrijednost val u polju A od n elemenata
auto retValue = cs.search(A, n, val);
```

Slijedno pretraživanje (korištenjem *strategije* - objektno oblikovnog obrasca)

Searching.cpp

```
template <typename T>
class LinearSearch : public ISearch<T> {
public:
   RetValSearch search(const T A[], const size_t n,
                                    const T& item) override {
      bool found = false;
      int index = -1;
      for (auto i = 0; i < n; i++) {
         if (A[i] == item) {
            found = true;
            index = i;
            break;
         }
      }
      return RetValSearch{found, index};
   }
};
```

Određivanje vremena izvođenja

- analiza algoritma za najlošiji slučaj
  - analiza za ulazni skup podataka za koji algoritam treba ***najdulje*** vrijeme izvođenja u odnosu na sve druge moguće ulazne skupove podataka
- analiza algoritma za najbolji slučaj
  - analiza za ulazni skup podataka za koji algoritam treba ***najkraće*** vrijeme izvođenja u odnosu na sve druge moguće ulazne skupove podataka
- obično nas zanima najlošiji slučaj
- prosječan slučaj se može posebno prikazati, ako se vrijeme izvođenja razlikuje od najlošijeg slučaja
- funkcija koja opisuje vrijeme izvođenja nekog algoritma u najboljem slučaju ne može asimptotski rasti brže od funkcije koja opisuje vrijeme izvođenja u najlošijem slučaju

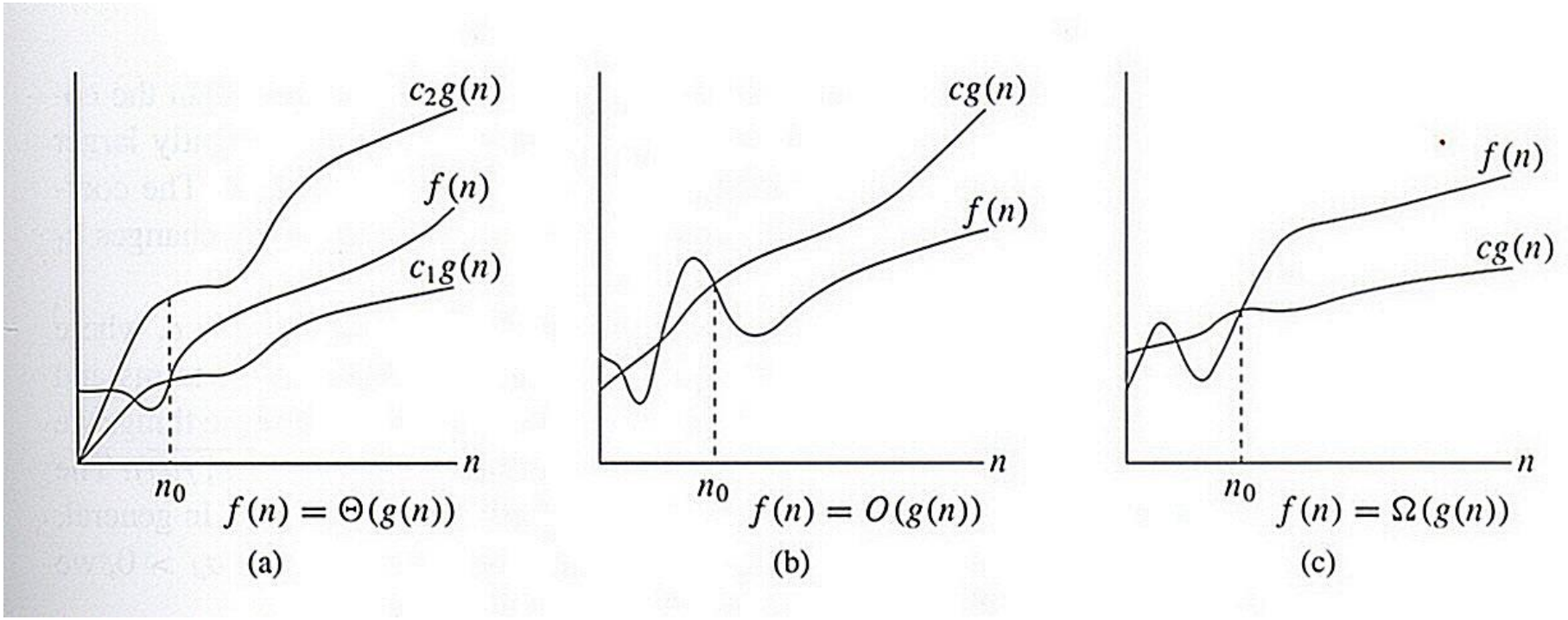
Asimptotska notacija (1)

- za bilježenje vremena izvođenja (još se naziva Bachmann–Landauova notacija; Landauova notacija)
  - simboli: O, Ω, Θ, o, ω
  - Paul Gustav Heinrich Bachmann (Berlin, 1837.– Weimar, 1920.) – uveo je *O* notaciju u knjizi *Analytische Zahlentheorie* („Analitička teorija brojeva”, 1894.)
  - Edmund Georg Hermann Landau (Berlin, 1877. – Berlin, 1938.) – popularizirao korištenje notacije
  - veliko *O* dolazi od *Ordnung von* (Bachmann, 1894.) – red veličine rasta funkcije (engl. *order of*)
  - Donald Knuth (Milwaukee, 1938. -) je 1970.-ih popularizirao *O* notaciju u računarstvu i uveo *Θ* notaciju

Asimptotska notacija (2)

- za bilježenje vremena izvođenja (simboli: O, Ω, Θ, o, ω)
  - uzima se u obzir samo **vodeći član funkcije *T*,** tj. najbrže rastući član
  - ne uzima se u obzir koeficijent uz najbrže rastući član
- $O(g(n))$ → asimptotska gornja granica za $T(n)$ (analogno ≤)
  - $T(n)$ asimptotski ne raste brže od $cg(n)$, $c > 0$
- $\Omega(g(n))$ → asimptotska donja granica za $T(n)$ (analogno ≥)
  - $T(n)$ asimptotski ne raste sporije od $cg(n)$, $c > 0$
- $\Theta(g(n))$ → asimptotska donja i gornja granica za $T(n)$ (analogno =)
  - $T(n)$ se nalazi između $c_1g(n)$ i $c_2g(n)$; $T(n)$ raste jednako brzo kao i $g(n)$

Asimptotska notacija (3)



Cormen, Leiserson & Rivest: *Introduction to algorithms*, 2009.

O – notacija (1)

- *O-notacija:* asimptotska gornja granica od $f(n)$ (engl. *asymptotic upper bound*)
- $f(n) = O(g(n))$ ako postoje dvije pozitivne konstante c i n_0 takve da vrijedi $0 \leq f(n) \leq c \cdot g(n)$ za sve $n \geq n_0$
- ILI:

  $O(g(n)) = \{f(n)$: postoje konstante $c > 0$ i $n_0 > 0$ takve da vrijedi $0 \leq f(n) \leq c \cdot g(n)$ za sve $n \geq n_0\}$
- $f(n) = O(g(n))$ znači $f(n) \in O(g(n))$
- drugim riječima, trajanje izvođenja procjenjuje se kao red veličine određen temeljem broja podataka n, pomnoženo s nekom konstantom c

O – notacija (2)

Primjer:

- $n^3+5n^2+77n = O(n^3)$
- vrijedi (po def.) isto tako i : $n^3 + 5n^2 + 77n = O(n^4)$, ali se obično bilježi najbliža gornja granica koja se može odrediti, tj. $O(n^3)$ u ovom slučaju

Primjer:

- Koliki je posao prenijeti 1 stolicu iz dvorane A u dvoranu B?
- Koliki je posao prenijeti *n* stolica iz A u B?
- Koliki je posao prenijeti *n* stolica iz A u B, s time da se kod donošenja svake nove stolice sve dotada donesene stolice u B moraju pomaknuti, pri čemu se za pomicanje stolice u dvorani B ulaže isti trud kao i kod prijenosa jedne stolice iz A u B?

$$1 + 2 + 3 + 4 + \ldots + n = n(n+1)/2 = n^2/2 + n/2 = O(n^2)$$

O – notacija (3)

- ako je broj izvođenja operacija nekog algoritma zadan polinomom m-tog stupnja i ovisan o nekom ulaznom argumentu n, onda je vrijeme izvođenja za taj algoritam $O(n^m)$ gdje $m = max\{m_i\}$
- Dokaz:

  Ako je vrijeme izvođenja određeno polinomom:

  $A(n) = a_m n^m + \ldots + a_1 n + a_0$

  $m = max\{m_i\}$

  onda vrijedi:

  $|A(n)| \leq |a_m| n^m + \ldots + |a_1| n + |a_0|$

  $|A(n)| \leq (|a_m| + |a_{m-1}|/ n + \ldots + |a_1| /n^{m-1} + |a_0|/ n^m) n^m$

  $|A(n)| \leq (|a_m| + \ldots + |a_1| + |a_0|) n^m$, za svaki $n \geq 1$

Odaberemo: $c = a_m + \ldots + a_1 + a_0$ i $n_0 = 1$, čime je tvrdnja je dokazana.

O – notacija (4)

- vrijedi za dovoljno veliki n:

  $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < \ldots < O(2^n) < O(n!)$

  - $O(1)$ znači da je vrijeme izvođenja ograničeno konstantom
  - ostale vrijednosti, do predzadnje, predstavljaju polinomna vremena izvođenja algoritma
    - svako sljedeće vrijeme izvođenja je veće za red veličine
  - predzadnji izraz $O(2^n)$ predstavlja eksponencijalno vrijeme izvođenja
    - ne postoji polinom koji bi ga mogao ograničiti, jer za dovoljno veliki n ova funkcija premašuje bilo koji polinom
  - algoritmi koji zahtijevaju eksponencijalno vrijeme mogu biti nerješivi u razumnom vremenu, bez obzira na brzinu slijednog računala

O – notacija – primjeri (1)

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < \ldots < O(2^n) < O(n!)$

- $O(1)$: ispitivanje je li broj paran ili neparan (*konstanta*)
- $O(\log n)$: binarno pretraživanje sortiranog polja (str. 49) (*logaritamska* funkcija)
- $O(n)$: ispis *n* članova polja (*linearna* funkcija)
- $O(n \log n)$: sortiranje polja od *n* članova korištenjem algoritma *merge sort* (sortiranje spajanjem)
- $O(n^2)$: ispis svih *n x n* članova polja kvadratne matrice; sortiranje odabirom (*selection sort*) (*kvadratna* funkcija)
- $O(n^3)$: jednostavno množenje kvadratnih matrica dimenzija *n* x *n* (*kubna* funkcija)

O – notacija – primjeri (2)

- $O(2^n)$: određivanje svih mogućih podskupova skupa koji ima n članova (*eksponencijalna* funkcija)
  - svaki član skupa S jest ili nije uključen u pojedini podskup, tj. dva su moguća stanja za svaki član skupa: 1 ili 0 (član/nije član podskupa skupa S)
  - svaki podskup možemo prikazati kao niz od n bitova: 0000…0 predstavlja prazan skup, a 111…1 sam skup S
  - svi se ostali podskupovi mogu generirati tako da se na prethodni podskup doda 1, tj.

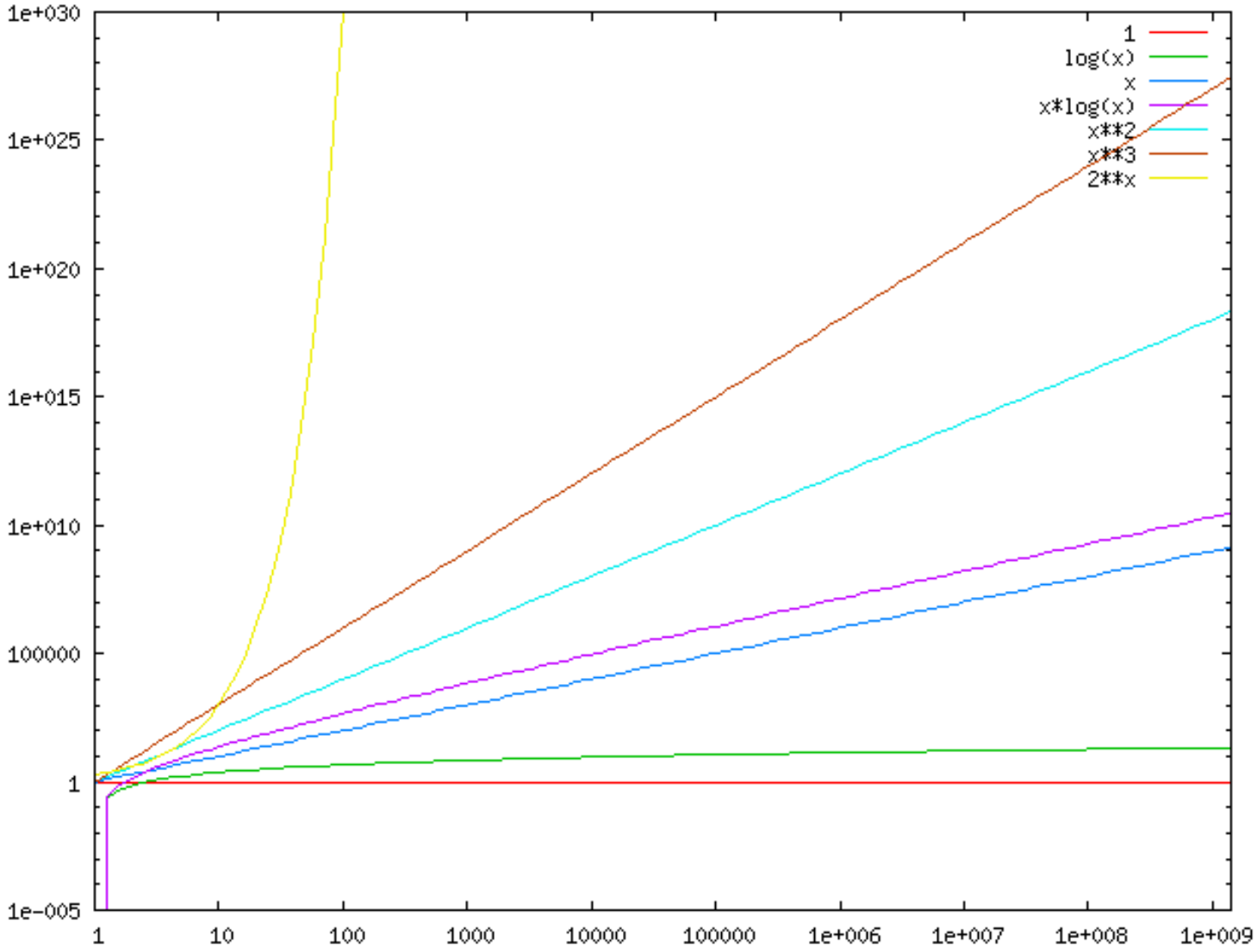
    000…00
    000…01
    000…10
    itd.
    111..11

O – notacija – primjeri (3)

- $O(n!)$: rješenje problema trgovačkog putnika isprobavanjem svih mogućih kombinacija (engl. *brute-force search*)
  - problem trgovačkog putnika: obići n gradova tako da ukupan put bude najkraći

  (kreće se od bilo kojeg od n gradova, zatim se odabire jedan od preostalih n-1 gradova, itd.)

Razne složenosti u logaritamskom mjerilu



Primjeri ovisnosti trajanja o broju podataka i složenosti

n	T(n) = $O(n)$	T(n) = $O(n \log(n))$	T(n) = $O(n^2)$	T(n) = $O(n^3)$	T(n) = $O(2^n)$
5	0.005 μs	0.01 μs	0.03 μs	0.13 μs	0.03 μs
10	0.01 μs	0.03 μs	0.1 μs	1 μs	1 μs
20	0.02 μs	0.09 μs	0.4 μs	8 μs	1 ms
50	0.05 μs	0.28 μs	2.5 μs	125 μs	13 dana
100	0.1 μs	0.66 μs	10 μs	1 ms	4×10^{13} godina

Što se može riješiti u zadanom vremenu

Vremenska složenost	Veličina najvećeg slučaja problema koji se može riješiti za 1 sat		
	Danas raspoloživim računalom	**100 puta bržim računalom**	**1000 puta bržim računalom**
n	N_1	$100 \cdot N_1$	$1000 \cdot N_1$
n^2	N_2	$10 \cdot N_2$	$31.6 \cdot N_2$
n^3	N_3	$4.64 \cdot N_3$	$10 \cdot N_3$
2^n	N_4	$N_4 + 6.64$	$N_4 + 9.97$
3^n	N_5	$N_5 + 4.19$	$N_5 + 6.29$

Relativni udio komponenti u izrazu:
$T(n) = n^4 + n^3 \log n + 2^{(n-1)} = O(2^n)$

n	n^4	$n^3 \log n$	$2^{(n-1)}$
1	1	0	1
10	10 000	1000	512
17	83 521	6 046	65536
18	104 976	7 321	131072
50	6 250 000	212 372	562 949 953 421 312
100	100 000 000	2 000 000	6.3×10^{29}

Utjecaj konstante $K = 1\,000\,000$

n	$K\,n$	$K\,n^2$	$K\,n^4$	$K\,2^n$
1	10^6	10^6	10^6	2×10^6
10	10^7	10^8	10^{10}	$1{,}0 \times 10^8$
100	10^8	10^{10}	10^{14}	$1{,}2 \times 10^{36}$
1 000	10^9	10^{12}	10^{18}	$\sim 10^{306}$

Ω - notacija

- Ω*-notacija*: asimptotska donja granica od $f(n)$ (engl. *asymptotic lower bound*),
  tj. izvođenje ne može trajati kraće od $\Omega(g(n))$
- $f(n) = \Omega(g(n))$ ako postoje dvije pozitivne konstante c i n_0 takve da vrijedi
  $f(n) \geq c \cdot g(n) \geq 0$ za sve $n > n_0$
- ILI:
  $\Omega(g(n)) = \{f(n)$: postoje konstante $c > 0$ i $n_0 > 0$ takve da vrijedi
  $0 \leq c \cdot g(n) \leq f(n)$ za sve $n \geq n_0\}$
- $f(n) = \Omega(g(n))$ znači $f(n) \in \Omega(g(n))$

O - notacija i Ω - notacija - primjeri

- ispis n brojeva: $\Omega(n)$ i $O(n)$
- množenje dviju matrica od n x n elemenata:
  - „školski" način: $O(n^3)$
  - postoje i brži algoritmi za množenje matrica:
    - Strassenov algoritam $O(n^{2.807})$ (Strassen, 1969.)
    - poopćenje Coppersmith–Vinogradovog algoritma $O(n^{2.373})$ (Le Gall, 2014.)
  - množenje matrica ne može biti brže od $\Omega(n^2)$, jer matrica reda n ima n^2 elemenata
    - to je donja granica problema, tj. vrijeme najbržeg mogućeg algoritma za rješavanje problema (u ovom slučaju za množenje matrica) ne može biti bolje od ove donje granice

Θ - notacija

- Θ-*notacija:* f i g rastu jednako brzo za velike n (engl. *asymptotic tight bound*)
- $f(n) = \Theta(g(n))$ ako postoje pozitivne konstante c_1, c_2 i n_0 takve da vrijedi $c_1 g(n) \leq f(n) \leq c_2 g(n)$ za sve $n > n_0$
- ILI:

  $\Theta(g(n)) = \{f(n)$: postoje konstante $c_1 > 0$, $c_2 > 0$ i $n_0 > 0$ tako da vrijedi $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ za sve $n \geq n_0\}$
- $f(n) = \Theta(g(n))$ znači $f(n) \in \Theta(g(n))$
- omjer f i g je između c_1 i c_2
- drugim riječima, ako vrijedi $f(n) = \Theta(g(n))$, onda vrijedi $f(n) = O(g(n))$ i $f(n) = \Omega(g(n))$

Θ - notacija – primjeri (1)

- ispis n članova polja: $\Omega(n) = O(n) = \Theta(n)$
  - za ispis n članova polja može se napisati i npr. $\Omega(1)$ i $O(n^2)$ (ispravno prema definiciji Ω i O),
    ali se obično odabiru $O(g(n))$ i $\Omega(h(n))$ tako da budu najbliža gornja i donja granica (eng. *tightest asymptotic upper/lower bound*)
  - dakle, u ovom je primjeru najprecizniji opis vremena izvođenja algoritma: $\Theta(n)$

- po abecedi poredati n kontrolnih zadaća tako da se prvo pronađe prvi po abecedi, zatim se u preostalima traži prvi itd. (zadaće inicijalno nisu poredane po abecedi)

$$O(n^2) = \Omega(n^2) = \Theta(n^2)$$

Θ - notacija – primjeri (2)

Searching.cpp

- **funkcija za ispis polja:** petlja se uvijek obavi n puta $\rightarrow \Theta(n)$
- funkcija **`trazi`**
  - u najboljem slučaju se u prvom koraku nađe traženi član polja: $\Theta(1)$
  - u najlošijem slučaju se trebaju pregledati svi članovi polja: $\Theta(n)$
  - u prosječnom se slučaju (slučajni brojevi) treba pregledati oko polovice svih članova polja: $\Theta(n)$
  - vrijeme izvođenja **algoritma** (uzimaju se u obzir svi slučajevi) pripada skupovima $\Omega(1)$ i $O(n)$
    - $\Omega(1)$ zato jer postoji skup podataka (najbolji slučaj) za koji se algoritam obavi u vremenu $\Theta(1)$
    - $O(n)$ zato jer postoji skup podataka (najlošiji i prosječan slučaj) za koji se algoritam obavi u vremenu $\Theta(n)$

Asimptotska notacija - napomene

- *O, Ω, Θ* – sve se notacije mogu primijeniti na najbolji, najlošiji i prosječan slučaj izvođenja nekog algoritma, ali i na algoritam u cijelosti
- ako kažemo da *vrijeme izvođenja nekog algoritma* pripada skupovima $O(n)$ i $\Omega(1)$, onda to znači da postoji neki skup ulaznih podataka za koji je vrijeme izvođenja $\Omega(1)$ i neki skup ulaznih podataka za koji je vrijeme izvođenja $O(n)$
- **Primjer:** slijedno pretraživanje u polju ***n*** nesortiranih članova
  - vrijeme izvođenja algoritma pripada skupovima $\Omega(1)$ i $O(n)$
  - za algoritam ne vrijedi $\Omega(n)$, zato jer postoji ulazni skup podataka za koji vrijedi $\Omega(1)$ (slučaj kada je traženi član na početku polja)

Zadatak (1)

- Pretpostavimo da su članovi polja **sortirani** (poredani uzlazno ili silazno).
- Kolike su složenosti u najboljem, najlošijem i prosječnom slučaju pri **slijednom** pretraživanju sortiranih zapisa?

```
Sortiraj članove polja
Ponavljaj za sve članove polja
    Ako je trenutni član polja jednak traženom
        Traženi član polja je pronađen
        Iskoči iz petlje
    Ako je trenutni član polja veći od traženog
        Član polja ne postoji
        Iskoči iz petlje
```

Zadatak (2)

- Odredite vrijeme izvođenja sljedećeg programskog odsječka u ovisnosti o *n*:

```
for (i = 1; i <= n; i++)
   for (j = 0; j < i; j++) {
      cout << i << " " << j << endl;
}
```

- Rješenje (zanemarujemo konstante i gledamo samo najbrže rastući član):

$T(n) = \sum_{i=1}^{n} \boldsymbol{i} = 1 + 2 + \ldots + n = n(n+1)/2 = \Theta(n^2)$

Zadatci za vježbu

Stvarno trajanje ($T(n)$)	O
$5n^2+7n+4$	n^2
$3n^2+700n+2$	n^2
$n^2+7\log n+40$	n^2
$0.1 \cdot 2^n+100n^2$	2^n
$(2n+1)^2$	n^2
$6n\log n + 10n$	$n\log n$
$8\log n + 1000$	$\log n$
$3n! + 2^n + n^{10}$	$n!$
$3 \cdot 5^n + 6 \cdot 2^n + n^{100}$	5^n

o - notacija

- $o(g(n))$ je asimptotska gornja granica od $f(n)$ *koja nije čvrsta*
- $o(g(n)) = \{f(n)$: za bilo koju konstantu $c > 0$, postoji $n_0 > 0$ tako da vrijedi $0 \leq f(n) < c \cdot g(n)$ za sve $n \geq n_0\}$
- *Značenje:* utjecaj $f(n)$ na vrijeme izvođenja postaje zanemariv u odnosu na funkciju $g(n)$ kako n raste, tj. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
- Primjer: $5n = o(n^2)$ i $5n = O(n)$, ali $\mathbf{5n \neq o(n)}$
- glavna razlika između *o* i *O* notacije:
  - ako vrijedi $f(n) = O(g(n))$, onda $0 \leq f(n) \leq c \cdot g(n)$ vrijedi za **neki** $c > 0$ za sve $n \geq n_0$
  - ako vrijedi $f(n) = o(g(n))$, onda $0 \leq f(n) < c \cdot g(n)$ vrijedi za **svaki** $c > 0$ za sve $n \geq n_0$

ω - notacija

- $\omega(g(n))$ je asimptotska donja granica od $f(n)$ *koja nije čvrsta*
- $\omega(g(n)) = \{f(n)$: za bilo koju konstantu $c > 0$, postoji $n_0 > 0$ tako da vrijedi $0 \leq cg(n) < f(n)$ za sve $n \geq n_0\}$
- *Značenje:* utjecaj $g(n)$ na vrijeme izvođenja postaje zanemariv u odnosu na funkciju $f(n)$ kako n raste, tj. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$
- Primjer: $5n^2 = \omega(n)$ i $5n = \Omega(n)$, ali $\mathbf{5n \neq \omega(n)}$
- glavna razlika između ω i Ω notacije:
  - ako vrijedi $f(n) = \Omega(g(n))$, onda $0 \leq cg(n) \leq f(n)$ vrijedi za **neki** $c > 0$ za sve $n \geq n_0$
  - ako vrijedi $f(n) = \omega(g(n))$, onda $0 \leq cg(n) < f(n)$ vrijedi za **svaki** $c > 0$ za sve $n \geq n_0$

Asimptotska notacija

- vrijeme izvođenja algoritma je $\Theta(g(n))$, ako je vrijeme izvođenja u najlošijem slučaju $O(g(n))$ i vrijeme izvođenja u najboljem slučaju $\Omega(g(n))$

- neka su $f(n)$ i $g(n)$ funkcije tako da vrijedi f: N → R^+, g: N → R^+. Onda vrijedi:
  - $O(f(n)) + O(g(n))$ je $O(f(n) + g(n))$
  - $O(f(n)) \cdot O(g(n))$ je $O(f(n) \cdot g(n))$
  - Ako je $f(n)$ polinom k-tog stupnja, onda $f(n) = O(n^k)$

Asimptotska notacija – primjeri

- $f(n) = n^2 + 2n \quad \in \Theta(n^2)$
- $f(n) = n^2 + 2n \quad \notin \Theta(n)$
- $f(n) = n^2 + 2n \quad \in \Omega(n)$
- $n \in O(n^2)$
- $\log_2 n \in O(\log_3 n)$
- $O(n^k) \subseteq O(n^l)$ za $0 \leq k < l$
- $O(f) = O(g)$ ako i samo ako $f \in O(g)$ i $g \in O(f)$
- $O(f) = O(g)$ ako i samo ako $f \in \Theta(g)$
- ako je $f_1 \in O(g_1)$ i $f_2 \in O(g_2)$ onda je $f_1 + f_2 \in O(\max\{g_1, g_2\})$
- ako je $f_1 \in O(g_1)$ i $f_2 \in O(g_2)$ onda je $f_1 \cdot f_2 \in O(g_1 \cdot g_2)$

Asimptotska notacija – zadatci

- Odredite asimptotsko vrijeme izvođenja u Θ-notaciji za sljedeće funkcije:
  - $\sum_{i=0}^{n} 2^i$
  - $\sum_{i=0}^{10} n^i$
  - n^n - 2^n

- Vrijede li sljedeće tvrdnje:
  - $2^{n+1} = O(2^n)$
  - $2^{2n} = O(2^n)$

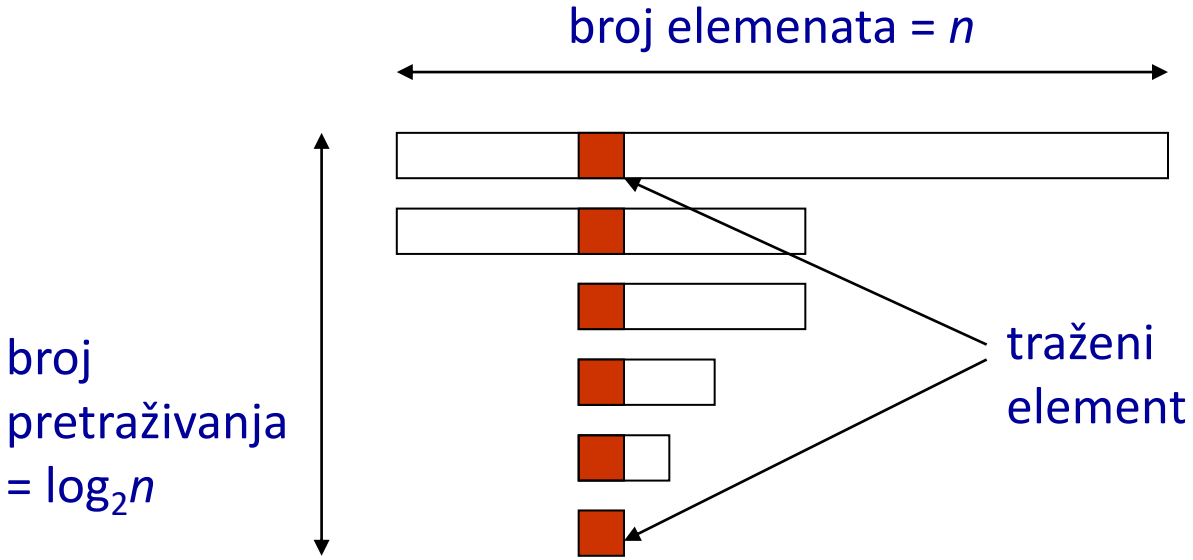
Određivanje prostorne složenosti u O, Ω, Θ – notaciji

- u praksi se određuje i prostorna složenost (engl. *space complexity*) nekog algoritma ovisno o veličini ulaznog skupa podataka
- Primjer:
  - prostorna složenost polja od n članova: $\Theta(n)$
    - ako pohranjujemo polje od n cijelih brojeva tipa `integer` (gdje svaki podatak zauzima 4 okteta; engl. *byte*), tada cijelo polje zauzima $4n$ okteta
  - prostorna složenost matrice dimenzija m x n: $\Theta(mn)$
- kod složenijih algoritama i velikih količina podataka često je potrebno odabrati između prostorne i vremenske složenosti (engl. *trade-off*), ovisno o tome koji je resurs ograničen ili kritičan

Binarno i blokovsko pretraživanje

Binarno pretraživanje

- preduvjet: podatci trebaju biti sortirani
- započinje na polovici polja (ili datoteke) i nastavlja se stalnim raspolavljanjem intervala u kojem bi se mogao naći traženi zapis
  - prosječni broj pretraživanja je $\log_2 n$ -1
- složenost je **O(log$_2$n)**



Primjer binarnog pretraživanja

- Tražimo broj 25

Algoritam za binarno pretraživanje

Searching.cpp

```
donja_granica = 0
gornja_granica = ukupan_broj_elemenata-1
Ponavljaj
    Izračunaj središnji element
    Ako je zapis u sredini jednak traženom
            Zapis je pronađen
            Iskoči iz petlje
    Ako je donja granica veća ili jednaka gornjoj
            Zapis nije pronađen
            Iskoči iz petlje
    Ako je zapis u sredini manji od traženog
            Postavi donju granicu na trenutni zapis + 1
    Ako je zapis u sredini veći od traženog
            Postavi gornju granicu na trenutni zapis - 1
```

Binarno pretraživanje
(korištenjem *strategije* – objektno oblikovnog obrasca)

Searching.cpp

```
template <typename T>
class BinarySearch : public ISearch<T> {
public:
   RetValSearch search(const T A[], size_t n,
                       const T& item) override {

      ...
```

Pitanja

1. Koja su vremena izvođenja u najboljem, najlošijem i prosječnom slučaju kod binarnog pretraživanja *n* zapisa?
2. Za slučaj abecedno poredanog popisa mjesta u Hrvatskoj (6935 mjesta), koliko najviše koraka treba napraviti kako bi se pronašlo traženo mjesto?
3. Hoće li binarno pretraživanje u svim slučajevima biti brže od slijednog (neovisno o veličini ulaznog skupa podataka)?

Zadatak

- Pretpostavite da se n podataka u nekom skupu može sortirati u vremenu $O(n \log_2 n)$.
- Podatci su na početku nesortirani.
- Trebate obaviti n pretraživanja u tom skupu podataka. Što je brže:
  - koristiti slijedno pretraživanje?
  - sortirati podatke i koristiti binarno pretraživanje?

Rješenje:

- ima više smisla sortirati i binarno pretraživati!
  - n slijednih pretraživanja nesortiranih podataka: $n \cdot O(n) = O(n^2)$
  - sortiranje + n binarnih pretraživanja:
    $O(n \log_2 n) + n \cdot O(\log_2 n) = O(n \log_2 n)$

Čitanje po blokovima

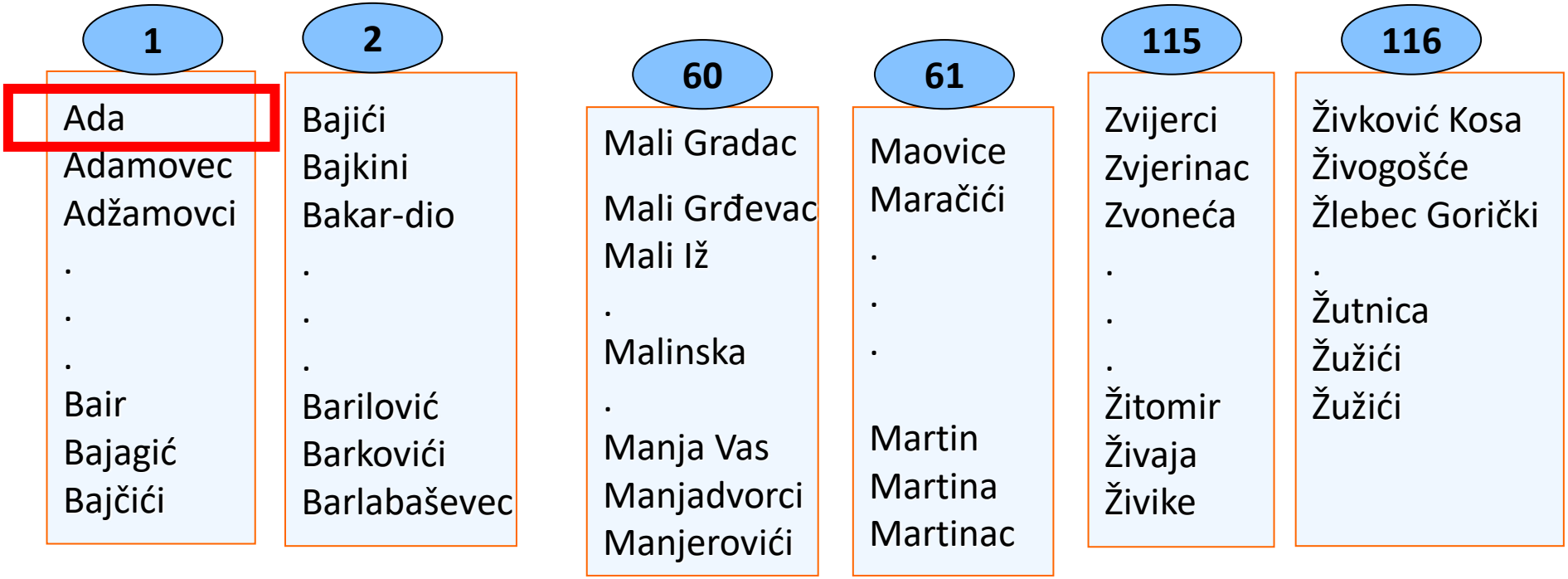
- Ideja čitanja po blokovima (engl. *jump search*; *block search*): kada su zapisi sortirani nije neophodno pregledavati sve zapise
  - može se pregledavati npr. svaki *B*-ti (npr. svaki stoti) zapis
  - u tom slučaju, kažemo da *B* zapisa čini jedan blok (pretinac)
  - kada se odredi položaj zapisa s traženim ključem, pripadni blok zapisa se slijedno pretraži

- Kako odrediti optimalnu veličinu bloka?

Primjer – popis mjesta u Hrvatskoj

- tražimo Malinsku u popisu od F = 6935 mjesta
- na svakoj stranici je navedeno B = 60 mjesta
- ima F / B vodećih zapisa (stranica): ⌈F / B⌉ = 116

Searching.cpp



Blokovsko pretraživanje
(korištenjem *strategije* – objektno oblikovnog obrasca)

Searching.cpp

```
template <typename T>
class JumpSearch : public ISearch<T> {
public:
   RetValSearch search(const T A[], size_t n,
                       const T& item) override {

      ...
```

Čitanje po blokovima: optimalna veličina bloka

- zadano je F stavaka, veličina bloka je B, vodećih zapisa je $\lceil F / B \rceil$
- pretraživanje vodećih zapisa po blokovima (slijedno):
  - očekujemo da će traženi vodeći zapis biti pronađen nakon prosječno (F / B) / 2 pročitanih zapisa
- pretraživanje unutar bloka (slijedno):
  - očekujemo da će se traženi zapis pronaći nakon prosječno B / 2 čitanja
- ukupni očekivani broj čitanja jest F / (2 B) + B / 2
- optimalna veličina bloka: $(F / (2\ B) + B / 2)'_B = 0 \rightarrow B = \sqrt{F}$
- ukupno vrijeme pretraživanja (prosječan i najlošiji slučaj):
  $\Theta(\sqrt{F}) + \Theta(\sqrt{F}) = \boldsymbol{\Theta(\sqrt{F})}$

Pitanja

- Kolika je optimalna veličina bloka za popis mjesta u Hrvatskoj?
  - F = 6935 mjesta
  - optimalna veličina bloka je: $B = \sqrt{F} = 83{,}27$

- Koji je način pretraživanja najbrži: slijedno, binarno ili blokovsko pretraživanje za *n* podataka smještenih u radnoj memoriji?
  - slijedno: $O(n)$
  - binarno: $O(\log_2 n)$
  - blokovsko: $O(\sqrt{n})$

Zadatak 1

Odredite vrijeme izvođenja u O, Ω i, ako je moguće, Θ notaciji za zadani programski isječak u ovisnosti o **m**. Polje **c** je polje cijelih brojeva čiji elementi mogu imati vrijednost **0 ili 1**.

```
s=0;
for (i=0; i<m; i++){
        if (c[i]==0){
                for (j=1; j<m; j=j*2)
                        s += A[i*m+j]*B[j];
        }
        else {
                for (j=0; j<m; j++)
                        for (k=j+1; k<j+5; k++)
                                s += A[k*m+i]*A[j*m+i];
        }
}
```

Zadatak 2

```
/* A je polje n cijelih brojeva (n >= 1).
 * Funkcija za sortiranje sortira niz
 * uzlazno i implementira algoritam
 * naveden u imenu funkcije.
 */

void f(int A[], int n) {
   // Funkcija napuniNiz generira
   // elemente niza u vremenu Θ(n)

   for (int i = n; i >= 1; i--) {
      napuniNiz(A, n);
      selectionSort(A, n); //Θ(n²)
      for (j = 0; j < n; j++)
         std::cout << A[j] << " ";
      std::cout << endl;
   }
}
```

Analiza *a posteriori*

- stvarno vrijeme potrebno za izvođenje na računalu

```
#include <iostream>
#include <chrono>
#include <ctime>
int main(void) {
   auto start = std::chrono::system_clock::now();
   // ...
   auto end = std::chrono::system_clock::now();
   std::chrono::duration<double> elapsed_seconds = end - start;

   std::time_t start_time = std::chrono::system_clock::to_time_t(start);
   std::time_t end_time = std::chrono::system_clock::to_time_t(end);
   std::cout << std::ctime(&start_time) << " " << std::ctime(&end_time)
             << " Elapsed time: " << elapsed_seconds.count() << "s";
}
```

Analiza *a posteriori*

- bolje je koristiti `steady_clock` (neosjetljiv na vanjske promjene sistemskog vremena) i učahuriti logiku u neki razred npr.

Aposteriori.cpp

```
#include <chrono>
#include <ctime>
using namespace std;
…
class StopWatch {
 private:
   chrono::time_point<chrono::steady_clock> start;
 public:
   void reset() { start = chrono::steady_clock::now(); }
   StopWatch() { reset(); }
   double elapsedSeconds() {
      chrono::duration<double> d = chrono::steady_clock::now() - start;
      return chrono::duration_cast<chrono::microseconds>(d).count() / 1000000.;
   }
};
```

Analiza *a posteriori*

- sad je program puno elegantniji

Aposteriori.cpp

```
…
int main(void) {
   StopWatch s;
...  // odsječak koji se mjeri
   cout << s.elapsedSeconds();
}
```

Analiza *a posteriori* - primjer