OSNOVNI ALGORITMI NA CIJELIM BROJEVIMA

Cilj danas: konstrukcija jednostavnih algoritama sastavljenih od:
- petlje (danas - samo jedne)
- uvjetnih naredbi

Naglasak na konstrukciji - pa sve radimo na najjednostavnijim podacima, a to su:
- cijeli brojevi (čak ćemo obično uzeti da su brojevi još i nenegativni, tj. iz $\mathbb{N} \cup \{0\}$).

- Napomena uz realizaciju u C-u:
  - deklaracija za tip je int ili unsigned int.
  - ako radimo s unosima, nećemo kontrolirati ulaz - da li pripada $\mathbb{N} \cup \{0\}$.
    Ta kontrola se lako doda, ili se uzme apsolutna vrijednost ulaza, na samom početku.

Počinjemo s "vanjacijama" na temu:
- znamenke broja $n \in \mathbb{N} \cup \{0\}$ u nekoj bazi b (≥ 2).

Ponovimo kako izgleda prikaz broja $n \in \mathbb{N}$ (paziti - $n > 0$) u zadnoj (fiksnoj) bazi $b \geq 2$.

$$n = a_k \cdot b^k + a_{k-1} \cdot b^{k-1} + \ldots + a_1 \cdot b + a_0$$

a_k: vodeća znamenka; a_0: najmanja (zadnja) znamenka

Uz ograničenje: $a_0, \ldots, a_k \in \{0, 1, \ldots, b-1\}$
i $a_k > 0$

ovakav prikaz je JEDINSTVEN $\Rightarrow$ ima smisla nešto "računati" s njim.

(Računanje s objektima koji ne moraju biti jedinstvenog oblika $\rightarrow$ neugodan / nemoguć problem zbog: IZBORA OBLIKA, tj. kojeg od mnogo izabrati (jednog, sve ?)
)

Broj znamenki broja $n \in \mathbb{N}$ u bazi $b \geq 2$ je (onaj $k+1$)

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

Prvi zadatak je baš taj problem:

Zadatak: Napisati program/algoritam koji nalazi broj znamenki broja n u bazi b.

Prvi korak u konstrukciji algoritma je:
"prepoznati ulaz i izlaz", tj
algoritam ima oblik (1. razina):

učitaj n, b
nađi što treba (broj znamenki)
napiši što se traži (broj znamenki)

Kod ovako jednostavnih algoritama, obično čak preskočim učitaj i napiši - jer dolaze na očita mjesta (početak i kraj), ali ih, općenito, ne smijemo zaboraviti.

Malo kasnije ću to preskakati, samo zato da se algoritam "u sredini" BOLJE VIDI. Ali, u programu, naravno, to mora pisati.

Što sad? Treba naći broj znamenki!
Kako?
To je lako - imamo formulu

$$\text{broj\_znamenki} = \lfloor \log_b n \rfloor + 1.$$

E sad oprez!!

Ono što radi u matematici, NE MORA raditi u aritmetici računala, posebno kad je riječ o REALNIM brojevima - zbog zaokruživanja.

Osim toga, i u cijelim (odn. prirodnim) brojevima treba voditi računa da je skup prikazivih brojeva u računalu KONAČAN.

Cilj – napraviti (sastaviti) algoritam tako da RADI za što veći ulaz (tj. što više mogućih ulaza)
– idealno = radi za svaki dozvoljeni (mogući) ulaz, tj. za sve pozitivne brojeve.

Sad se sjetimo da u realnoj aritmetici imamo greške zaokruživanja i pogledajmo da li je to opasno u našoj formuli

$$\lfloor \log_b n \rfloor + 1$$

ovo je realan broj.

Funkcija $\lfloor\ \rfloor$ je OSJETLJIVA NA GREŠKE – posebno tamo gdje IMA SKOKOVE – u brojevima

$$n = b^k$$

Ako se $\log_b n$ izračuna malo pogrešno na krivu stranu, a to znači da je

"izračunati $\log_b b^k$" malo manji od k,

onda će najveće cijelo od toga "promašiti" za 1.
(Usput, očekujem da $\log_b n$ NEĆE pogriješiti za 1 ili više)

Mogući primjer – ako je $b=10$, $n=100$ i dobijemo

$$\log_{10} 100 = 1.99\ldots.98\ldots.$$

(na pr. u double, a ovih 8 je 17. znamenka).
Posljedica: broj znamenki je 2 (krivo), a ne 3.

Razlog: $\log_b$ (a C ima samo $\log = \ln$ i $\log 10 = \log$)
se računa otprilike kao:

$$\log_b n = \left\{ \frac{\log}{\log 10} \right\}(n) \Big/ \left\{ \frac{\log}{\log 10} \right\}(b)$$

i još se funkcija log ili log10 — računa nekom aproksimacijom (procesor ili matematička biblioteka)
Dakle – mogućnosti za grešku zaokruživanja ima dosta ⇒ OPREZ!

Test "opasnih" točaka:

- uzmem par razuih b-ora:
  na pr. $b=2$, $b=10$, $b=3$
- računam $\log_b(n)$ za $n=b^k$
  s tim da variram k tako da n bude ponkazir
  (Rezultat je double – u C-u)
- napravim ⌊ ⌋ od toga (# pretvaranje tipova (int) rezultat – u C-u)
- provjerim da li je konačni rezultat $=k$.

Pokaži: TLOG2 za $b=2$ → RADI!
(Zašto radi??? – koristim funkciju log, što je $\log_e = \ln$, pa kako onda radi?)
TLOG10 10 za $b=10$, preko log 10 → RADI!
(začudo!)

Međutim: TLOG10 za $b=10$, ali preko log } NE RADI!
i TLOG3 za $b=3$, preko log

Pouka: Izbjegavati "realnu" aritmetiku za cjelobrojne probleme – kad god je to moguće (i ne traje predugo!)
Inače – oprez → treba dosta znati o greškama zaokruživanja, pa ih (katkad) možemo "kontrolirati".
Probajte staviti $n+0.5$ (ili neki sličan pomak, ali <1).

Dakle, pustimo trenutno realnu aritmetiku "na miru" (tako je "brza", ali može pogriješiti!)

Idemo napraviti prave cjelobrojne algoritme i vidjeti koliko to "košta".

Kako doći do algoritma?

Prvi pokušaj (opet "prečicom"):

Iz $n = a_k b^k + \ldots + a_0$ vidim ovo:

Ako je broj znamenki $= k+1$, onda je

$$n \geq b^k \quad (\text{zbog } a_k > 0)$$

i

$$n < b^{k+1} \quad (\text{zbog } a_0, \ldots, a_k < b).$$

Ideja: "dižem" potenciju (eksponent) baze, sve dok ne prijeđem n

(eksponent je $k+1$ = broj znamenki, potencija je b^{k+1}).

Na pr., ovako:

Alg. 1

```
pot = 1;   (baza°) tj (b°)
broj_znamenki = ∅;   (eksponent)
sve dok je pot ≤ n ponavljaj
    [ pot = pot * b;
      broj_znamenki = broj_znamenki + 1;
```

(Prije toga učitam n, b. Iza – ispišem broj_znamenki).

Pitanja:

- Da li ovo radi (u principu – ne mora na računalu) za <u>svaki</u> prirodni broj n?
  (Kako to provjeriti / testirati – koji primjeri su dobri?)
  (Kako to dokazati?!)
- Što radi za $n = \emptyset$? (Vrati broj_znamenki $= \emptyset$, što ima nekog smisla, jer $n = 0$ <u>ne</u> ide u raniju tvrdnju o pokazu, zato jer vodeća znamenka NE MOŽE biti > 0).

Tj. ako NE brojim VODEĆE NULE, onda ispada da $n = 0$ zaista ima $\emptyset$ znamenki!

– Sasvim pošteno, rezultat za $n=0$ je pitanje DOGOVORA
(Ako hoću rezultat 1, onda treba dodati test!)

Druga varijanta:

Ideja – stvarno brojim znamenke, jednu po jednu.

Do zadnje znamenke a_0 je lako doći (Euklidov tm.)

$$a_0 = n \bmod b \quad (\text{mod} \to \% \text{ u C-u}).$$

Što onda? Kad napravim s tom znamenkom sve što treba (tj. povećam brojač za 1), onda ju OBRIŠEM iz broja.

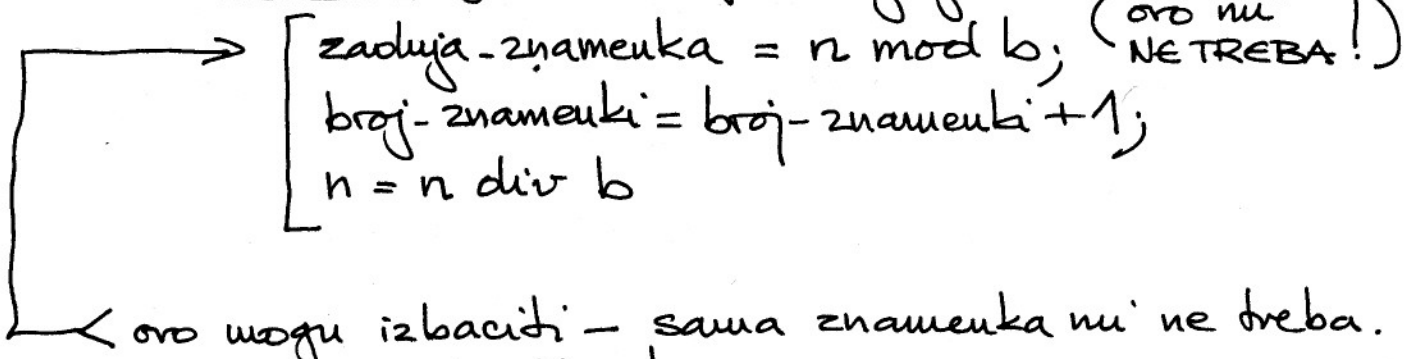
Kako? Tako da n cjelobrojno podijelim s b (div!).

$$n \text{ div } b = \begin{cases} 0 & \text{za } n < b \\ a_k \cdot b^{k-1} + \ldots + a_2 \cdot b + a_1, & \text{za } n \geq b \end{cases}$$

Ono spremim u n. Ako je to >0, onda je a_1 zadnja znamenka, pa ponovim "mod, div".
I tako to, sve dok ne dobijem nulu!

Alg. 2: broj-znamenki = ∅; (inicijalizacija na ∅ = još ništa nismo izbrojali!)

sve dok je $n > \emptyset$ ponavljaj



– Ista pitanja:
– Da li ovo radi za $\forall n \in \mathbb{N}$? (u principu). Dokaz!?
– Za $n=0$? (rezultat = ?)

Krasno! Sad imam 2 algoritma.
(Što će mi dva? Jedan je dovoljan! Ali; koji?)

Koje su bitne razlike između ova dva algoritma?

(1.) Prvi ostavi n "na miru" (ono što smo učitali ostaje u n i na kraju)
Drugi je "destruktivan" – uništava n i na kraju je sigurno $n = 0$ (ako je na ulazu bilo $n \geq 0$).
(Zašto? Odgovor – izlaz iz "while" – sve dok može samo s $n = 0$ (odn. $n \leq 0$)).

Ako nam n kasnije treba za još nešto, onda ga je zgodno iskopirati na početku u neku privremenu lokaciju i tamo "uništiti"

temp_n = n

i algoritam ide na temp_n (a ne na "n").

Međutim, ovo nije jako bitno!

(2.) Trajanje (složenost, brzina)?
Što se "broji"?
Elementarne operacije – na cijelim brojevima.
U oba algoritma imam:
- unutar petlje – dižem brojač
  - jednom množim, drugi put mod
  (dakle, pojedini prolazi su usporedivi po složenosti)
- broj prolaza – svaki prolaz "broji" po jednu znamenku (pažljivo pogledati!)

Dakle, # broj prolaza je JEDNAK u oba algoritma
⇒ imaju "podjednaku" složenost
(isti broj elementarnih operacija).

Tj. tu nema bitne razlike!

(Složenost logaritamski ovisi o n → v. kasnije
za zapis $T(n) = O(\log n)$).

(3.) Ključna razlika = korektnost u aritmetici računala!

(a baš to je bitno u praksi – kad napišemo program)

Dakle, gledamo za koje prikazive brojeve (u računalu) algoritmi rade korektno!

Tj. pretpostavimo da su ulazni brojevi n, b prikazivi!

(Ha, a što ako nisu? – Probajte!
Npr. $n = 10^{12} = 1000000000000$ u 32-bitnoj aritmetici)

Neće ići! Uvijek su prikazivi – bar ono što učitamo i spremimo u lokacije za n, b (iako to NE MORA biti ono što smo zaista NAPISALI na ulazu).

Alg. 1 – diže potenciju baze dok ne prijeđete n.

⇒ ta potencija raste i (egzaktno) može postati neprikaziva (tj. u aritmetici računala dobijemo neki rezultat – modulo $2^{\text{broj-bitova}}$, ali taj više NIJE korektan)

⇒ Alg. 1 može i "krepati" (oprez, čim nešto raste u cjelobrojnoj aritmetici)

Nažalost, zaista može!

Npr. – u 16-bitnoj aritmetici (max = 32767)
uz bazu $b = 10$, dobijemo

$$10^5 = \underbrace{100\,000}_{\text{pot. više prikaziv.}} > 32767$$

Ovo se događa za sve $n \geq 10000 = 10^4$, tj. Alg. 1 radi za manje od 1/3 prikazivih brojeva. Užas!

U 32-bitnoj aritmetici – nije puno bolje
(max = 2 147 483 647)
Strah ne radi za sve 10-znamenkaste brojeve, a duh je preko pola:

$$10^{10} = 10000000000 > 2\,147\,483\,647$$

Ispada da Alg.1 ne mijedi "u pišljivog boba".

Za razliku od toga, Alg. 2 SMANJUJE n (jer div i mod rade korektno za pozitivne brojeve). Tj. svi rezultati (međurezultati) u algoritmu su pozitivni (i korektni)

⇒ Alg 2 radi korektno za sve pozitivne brojeve.

Dakle, Alg. 2 je BOLJI.

—·—

Alg. 2 je ujedno i dobra podloga za sve ostale algoritme koji nešto rade sa svim znamenkama broja.

Jest – izdvaja znamenke "straga" (obratno od našeg čitanja/pisanja).

Probajte to napraviti "sprijeda" – ali tako da uvijek radi.

Dosad smo "brojali" znamenke. A sad, varijacije na temu "znamenke" – zbroj (suma), produkt, najveća, najmanja, pa "tražење" (postoji, svaka).

—·—

Osnovni cilj – korektne inicijalizacije objekata.

Općenito: zbroj (što uključuje i brojanje = zbrajaj po 1)
produkt (i sl.)

~~[redacted]~~ → inicijalizacija se radi na NEUTRALNI element (ako ga ima) i to PRIJE početka bilo kakve obrade

Dakle: suma se inicijalizira na $\emptyset$
produkt se inicijalizira na 1.

Pozadina = dogovor o poraštanju $\sum$, $\prod$ na praznom skupu (a to je ono "prije početka obrade")

$$\sum_{i=1}^{n} x_i = 0, \text{ za } n<1; \quad \prod_{i=1}^{n} x_i = 1, \text{ za } n<1.$$

(To je dogovor – zato oprez – pogledati da li takva inicijalizacija ima smisla)

Za operacije "bez neutrala" na skupu (nizu) podataka, ovo NE IDE. Tipični primjer su min, max.

Tada nema inicijalizacije prije početka obrade (tj. na praznom skupu) – jer min, max praznog skupa NISU definirani.

Inicijalizaciju takvih operacija možemo korektno napraviti tek kad naiđemo prvi element dozvoljenog skupa.

Izbjegavajte tzv. "lažne inicijalizacije" za min, max na neki podatak koji se ne može pojaviti u "pravom" skupu ili nizu.

Zbroj znamenki = zbrajam redom, jednu po jednu!

```
zbroj = 0;
sve dok je n > 0 ponavljaj
    [ zbroj = zbroj + n mod b;   ← obradi znamenku
      n = n div b;               ← obriši znamenku
                  (n mod b: znamenka)
```

Produkt znamenki:

```
prod = 1;
sve dok je n > 0 ponavljaj
    [ prod = prod * (n mod b);   ← obradi
      n = n div b;               ← obriši
```

Ako dozvolimo n = 0, ovo baš i nije pametno!
Zato oko svega ide test:

ako je n > 0 onda
    naš algoritam i rezultat
inače
    piši poruku!

Uočiti da je ovaj test n > 0 ⇔ n ima bar jednu znamenku u bazi b

pa je dobra podloga za korektne inicijalizacije!

Najveća znamenka:
- inicijalizacija na neku (najlakše - na zadnju)
- diži maksimum "ako treba"

```
ako je n>0 onda
  [max = n mod b;   {zadnja}
  n = n div b;      {obriši}
  sve dok je n>0 ponavljaj
     [znam = n mod b;
     ako je znam > max onda } diži max
        [max = znam;       } ako treba
     n = n div b;
  napiši max
inače
  [napiši poruku   (mogli smo pustiti i max = ∅)
```

Cijela stvar bi radila i "kraće", uz lažnu inicijalizaciju max = −1 (ispod dozvoljenih podataka - znamenki, jer dižemo maksimum).
Ali, "nije lijepo" i vodi u propast, kad nema "ispod".

Najmanja znamenka:
- spuštaj minimum, ako treba!

```
ako je n>0 onda
  [min = n mod b;
  n = n div b;
  sve dok je n>0 ponavljaj
     [znam = n mod b
     ako je znam < min onda } spusti min
        [min = znam        } ako treba
     n = n div b;
  napiši min;
inače
  [napiši poruku;
```

Što bi ovdje radilo kao lažna inicijalizacija?
(min = ... ?)

Za min i max smo odmah mogli ustanoviti da li ima "dozvoljenih" objekata - jer je svaka znamenka "dozvoljena" - pa test $n > \emptyset$ odmah dozvoljava inicijalizaciju na zadnju znamenku.
recimo

Kod malo složenijih operacija, to ne mora biti tako jednostavno - potraga za prvim dozvoljenim objektom se može pretvoriti u "pravu petlju".

Zadatak Naći najmanju (najveću) parnu znamenku (ako takva postoji).

Algoritam - oprez: nema inicijalizacije dok ne nađemo PRVU parnu znamenku!

- oprez 2: takva NE MORA POSTOJATI!

Ovo je klasična situacija za pretragu (traženje):

traži dok ne nađeš ili više nema smisla tražiti
prvi uvjet drugi uvjet.

I, u principu, SVE PRETRAGE IMAJU OVAJ OBLIK.
Izuzetak - kad ste SIGURNI da ćete naći ono što tražite - drugi uvjet može se IZBACITI.

Provjere i "kvantifikatori"
- da li postoji objekt sa zadanim svojstvom? (∃)
- da li svaki objekt ima zadano svojstvo? (∀)

- Rezultat je logičkog tipa (odgovor DA/NE, ISTINA/LAŽ)
- Realizacija: inicijalizacija odgovora
  sve dok nisi prošao sve objekte
    obradi sljedeći objekt

obradi sljedeći objekt = provjeri da li on ima traženo svojstvo i "popravi odgovor"

popravi odgovor: ∃ → vezmik ili
  odgovor = odgovor ili "oraj dobar"
∀ → vezmik i
  odgovor = odgovor i "oraj dobar"

inicijalizacija - oprez:
za ∃: na laž (jedan ili pretrara u istinu)
∀: na istina (-11- i -11- u laž)

⇐ u praznom skupu NE postoji "dobar"
—— 11 —— su SVI "dobri" (jer ih NEMA!)

- Pretraga se može SKRATITI, ovisno o odgovoru, tj. ne trebam proći sve objekte, nego ranije saznam odgovor:
  za ∃: stanem kad nađem prvog (čim odgovor postane istinit)
  za ∀: stanem kad nađem prvog koji nije dobar (čim odgovor postane lažan)

tj. za ∃: tražim dok je odgovor [redacted] LAŽAN (kao u inic.)
za ∀: —— 11 —— ISTINIT (-11-)

i, naravno, pazim da ima objekata za "pretragu".