O predstavitvi podatkov v računalniku: cela števila

↓↓↓
Jure Slak

→ **Velikokrat slišimo, da računalniki vse podatke hranijo kot zaporedja ničel in enic. Vse lepo in prav, to za moderne računalnike drži, vendar ne pove prav veliko o tem, kako so shranjeni konkretni podatki. Večina podatkov, ki jih shranjujemo, npr. slike, zvok ali besedilo, so shranjena kot zaporedja celih števil.**

Obstaja več dogovorjenih načinov, kako sliko, zvok ali besedilo pretvorimo v zaporedje števil. Ti načini so večini ljudi znani kot datotečni formati, za slike poznamo `jpg`, `png`, `gif` (in druge), za zvok `mp3`, `flac` ipd. Običajni uporabniki ne potrebujejo nobenega znanja o tem, kako formati delujejo, pomembno je le, da oba programa, tako tisti, ki shranjuje, kot tisti, ki prikazuje, uporabljata enak format. S predstavitvijo večjih kosov podatkov, kot so slike ali besedilo, se bomo ukvarjali kdaj drugič; tokrat pa se bomo spustili še en nivo globlje in se vprašali, kako delamo s seznamom števil oziroma celo kako predstavimo le eno celo število. Morda je odgovor na prvi pogled preprost: pretvorimo število v dvojiški sistem, pa smo! To je tudi res, vendar bo po poti potrebno rešiti še nekaj težav.

Dvojiški sistem

Kot vam je verjetno znano, števila običajno zapisujemo v desetiškem sistemu in za zapis števil uporabljamo števke od 0 do 9. Število 593 tako pomeni

- $593 = 5 \cdot 100 + 9 \cdot 10 + 3 = 5 \cdot 10^2 + 9 \cdot 10^1 + 3 \cdot 10^0$.

Podobno lahko vsako število zapišemo v kakšnem drugem sistemu, za računalništvo je posebej zanimiv dvojiški, saj ima le dve števki: 0 in 1. Število 593 bi tako napisali kot

- $$\begin{aligned} 593 &= 1 \cdot 512 + 0 \cdot 256 + 0 \cdot 128 + 1 \cdot 64 + 0 \cdot 32 + \\ &\quad + 1 \cdot 16 + 0 \cdot 8 + 0 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 \\ &= 1 \cdot 2^9 + 0 \cdot 2^8 + 0 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + \\ &\quad + 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ &= 1001010001_{(2)}, \end{aligned}$$

kjer s podpisano (2) označimo, v katerem sistemu je število zapisano (v desetiškem podnapis ponavadi izpustimo).

Drugi zanimivi sistemi so potence dvojiškega, npr. osmiški in šestnajstiški, saj dovolijo krajši zapis kot dvojiški, vendar jih je še vedno enostavno pretvoriti v dvojiškega ali nazaj; enostavno pretvorimo vsako števko posebej v ustrezno število bitov (običajno bi morali uporabiti npr. Evklidov algoritem). Zgornje število bi v osmiškem zapisali kot $1121_{(8)}$.

Sistemske omejitve

Osnovna enota informacije je en bit – vrednost, ki je lahko le 0 ali 1. Toda, računalniki s posameznim bitom naenkrat ne morejo delati neposredno. Najmanjša enota, ki jo lahko neposredno uporabimo, je en bajt – ta ima osem bitov. Računalniški pomnilnik (RAM) si lahko predstavljamo kot ogromno tabelo s prostori, ki so veliki en bajt. Če imamo npr. 4GiB[1] pomnilnika, imamo torej prostora za 4 294 967 296

[1]Formalno obstaja razlika med enotama GiB in GB. Enote brez ›i‹, kot so kB, MB, GB, uporabljajo standardne fizikalne predpone kilo-, mega-, giga- in faktor 1000, medtem ko enote z ›i‹, torej KiB, MiB, GiB, uporabljajo predpone kibi-, mebi-, gibi- in faktor 1024. V praksi se uporablja zmešnjava vsega. Več o standardizaciji lahko preberete na strani ameriškega nacionalnega inštituta za standarde in tehnologijo: `physics.nist.gov/cuu/Units/binary.html`

bajtov. Vsak bajt predstavimo kot zaporedje osem bitov in ga shranimo v posamezno celico, kot je shematsko prikazano na sliki 1.

naslov	podatek
0	10101001
1	10110001
2	01011000
3	10100011
4	00001001
⋮	
4294967294	01000000
4294967295	10011110

SLIKA 1.
Shema računalniškega polnilnika

Sedaj ko poznamo omejitve sistema, se lahko pogovorimo o tem, kako velika števila bomo shranjevali. Nekateri programski jeziki, npr. Python, dopuščajo uporabo poljubno velikih števil. Vendar je to zgolj iluzija za uporabnika: ko števila postanejo prevelika, se v ozadju shranijo kot zaporedje manjših števil. Če dopuščamo le omejena števila, je najbolj skopuška možnost, da bi omejili delo s števili na en bajt. Največje število, ki bi ga tako lahko zapisali, bi bilo $11111111_{(2)}$ oz. 255. To bi bilo za vsakodnevno uporabo premalo, saj si hitro lahko zamislimo uporabo za večja števila. Običajno imamo pri programiranju opravka s števili, ki so velika 32 ali pa 64 bitov (torej štiri ali osem bajtov). Največje število, ki ga lahko shranimo v 32 bitov, je $11111111111111111111111111111111_{(2)}$ oziroma $2^{32}-1 = 4\,294\,967\,295$, kar je malo več kot štiri milijarde. Največje 64-bitno število, pa je $2^{64}-1 = 18\,446\,744\,073\,709\,551\,615$, kar je nekaj več kot 18 trilijonov. Že 32 bitna števila so dovolj za večino praktičnih potreb, vendar so kdaj neustrezna. Naštejmo nekaj znanih težav, kjer 32-bitna števila niso bila zadostna. Ponavadi imamo opravka s predznačenimi števili, ki lahko hranijo pozitivne in negativne vrednosti, tako da se zgornja meja zmanjša za približno polovico, na 2 147 483 647. Pred nekaj leti je YouTube video Gangam style dosegel dovolj ogledov in so morali število ogledov hraniti v 64-bitnih številih. Za težavami z omejenimi števili trpijo tudi marsikatere (pogosto starejše) računalniške igrice. Igra *Minecraft* je do verzije 1.7.3 napačno generirala teren, če je igralec šel predaleč od sredine sveta (ta teren je postal znan pod imenom Daljne dežele (*Farlands*)), in še danes imajo nekatere verzije to težavo. Zanimiva težava, ki nas še čaka, pa je povezana s shranjevanjem časa. Računalniki pogosto shranjujejo trenutni čas kot število sekund od 1. januarja 1970 in v torek 19. januarja 2038 bo število sekund preveliko, da bi ga shranili v 32-bitno število. Ta datum je znan kot »konec časa« in do takrat morajo programi začeti uporabljati 64-bitna števila. Pri shranjevanju časa bodo 64-bitna števila zadostovala še za naslednjih 585 milijard let in nas pred tem lahko skrbi marsikaj drugega.

Kljub temu, da kdaj povzročajo težave, so omejena števila praktično neizmerno uporabna. Če se dogovorimo, da bomo uporabljali 32-bitna števila, vemo, da moramo prebrati naslednje štiri bajte, in ti vsebujejo naše število. V nasprotnem primeru bi morali na kakšen bolj zapleten sporočiti, kako dolgo je število (morda najprej poslati dolžino, toda tudi to je število, kako pa sporočimo število). V praksi se pri večini zapisov in protokolov dogovorimo za omejitev velikosti števil.

Veliki in mali endian

Denimo, da želimo zapisati ali poslati število 2 740 498 857 z bitno reprezentacijo

- 2 740 498 857 =
  10100011 01011000 10110001 10101001,

kjer smo števke s presledki ločili po bajtih. Za lažje razumevanje jih oštevilčimo:

1. bajt	2. bajt	3. bajt	4. bajt
10100011	01011000	10110001	10101001

Pojavljata se dva smiselna načina pošiljanja: ali bajte pošljemo v vrstnem redu 1234 ali pa 4321. Bomo najprej poslali bajt 1, ki vsebuje »največje« števke (t. i. *most significant byte*) ali četrti bajt z »najmanjšimi« števkami? V praksi glede tega ni bilo sklenjenega kompromisa in uporabljata se oba načina. Pri komunikaciji prek omrežja se bajte ponavadi pošilja v vrstnem redu 1234, torej najprej »z velikega konca«, pri zapisu v pomnilnik ali pri procesorskih operacijah pa se na večini modernih arhitektur uporablja zapis 4321, torej »z manjšega konca«. Zgornje

število je v pomnilniku na sliki 1 zapisano z manjšega konca v celicah 0–3. Prikaz obeh zapisov je bolj pregledno napisan na sliki 2.

mali endian

naslov	podatek	
x	10101001	4. bajt
$x+1$	10110001	3. bajt
$x+2$	01011000	2. bajt
$x+3$	10100011	1. bajt

veliki endian

naslov	podatek	
x	10100011	1. bajt
$x+1$	01011000	2. bajt
$x+2$	10110001	3. bajt
$x+3$	10101001	4. bajt

SLIKA 2.
Mali in veliki endian zapis števila 2 740 498 857

Zapis števila 4 kot 32-bitnega števila z »manjšega konca« bi bil tako

- $4 = 00\ldots0100 = \begin{matrix}00000100\\00000000\\00000000\\00000000\end{matrix}$.

Angleški izraz za zapis »z velikega konca«, torej 1234, je *big endian*, za zapis »z manjšega konca«, torej 4321, pa *little endian*. Izraza izhajata iz satiričnega romana Guliverjeva potovanja, ki ga je napisal Johnatan Swift leta 1726. Roman je kritika takratne angleške družbe; v njem Guliver doživi brodolom in se zbudi v deželi Liliput, kjer živijo Liliputanci. Deželo težijo razdori in borbe med Liliputanci, eden izmed glavnih razlogov pa je, ali je treba trdo kuhano jajce najprej razbiti na zgornji (manjši) ali spodnji (večji) strani. V angleščini so se zagovorniki strani, ki je jajce razbijala z manjšega konca (*from the little end*) imenovali *little endians*, njihovi nasprotniki pa *big endians*. Izraz je v računalništvu popularizi-ral Danny Cohen v malo manj, a kljub vsemu nezanemarljivo satiričnem članku *O svetih vojnah in prošnja za mir* [1], objavljenem s strani takratne »delovne skupine za internet« (*Internet Engineering Task Force (IETF)*). Tam potegne nekaj vzporednic med zagovorniki ene in druge strani in jih tudi okliče za *Little-* ali *Big-Endians*. Od takrat se je izraz obdržal, splošnemu principu, kako lahko števila različno napišemo, pa rečemo *endianess*. V slovenskem prevodu se zgornji in spodnji del jajca pri Liliputancih imenujeta »vršek« in »tušek«, vendar terminologija (še?) ni prišla v rabo pri računalniških arhitekturah.

Negativna števila

Do sedaj smo se ukvarjali samo z zapisom pozitivnih števil. Običajna števila, ki jih uporablja računalnik, so predznačena, torej imajo predznak + ali −, in so lahko negativna. Kako pa zapišemo ta? En način je, da prvi bit žrtvujemo za predznak: dogovorimo se, da če je prvi bit enak 0, pomeni, da je število pozitivno, sicer pa negativno, preostanek pa interpretiramo kot običajno pozitivno število. Tako bi npr. 00000001 pomenilo število 1, 10000001 pa −1. Ta sistem se imenuje »predznak in velikost«, *sign and magnitude*

Na žalost se ta način ne uporablja pri celih številih. Zgoraj opisani način ima med drugim težavo, da ima dve ničli (+0 in −0), ki sta kot števili seveda enaki, bitna zapisa imata pa različna. To povzroča tudi težave pri postopku za seštevanje. Sistem, ki se večinoma uporablja dandanes, se imenuje »komplement dvojke« (*two's complement*) in teh težav nima, ima pa tudi smiselno matematično strukturo.

Da bo manj pisanja, denimo, da delamo z osem bitnimi števili. Pa se vprašajmo, katero število je −5? To je število, ki ga moramo prišteti 5, da dobimo 0. Trdim, da je pri osem bitnih številih to število v resnici 251. Pa poglejmo: 5 v dvojiškem zapišemo kot $101_{(2)}$, 251 pa kot $11111011_{(2)}$. Pisno ju seštejmo:

- $\begin{array}{cc|ll} & & 00000101_{(2)} & = 5 \\ + & & 11111011_{(2)} & = 251 \\ \hline = & 1 & 00000000_{(2)} & = 256 \end{array}$

Ker imamo samo osem bitov, rezultatu »odpade« prvi bit in dobimo odgovor 0. Če torej velja, da je $5 + 251 = 0$, je 251 nasprotna vrednost 5, torej −5. Po enakem principu je tudi $5 = -251$. Stvar dogovora je, kako si bomo posamezna števila interpretirali: če se dogovorimo, da binarni zapis 251 ne pomeni 251 ampak −5, potem pač nimamo nobenega načina, da bi zapisali število 251. Ampak nič hudega, jasno je, da bomo morali nekaj števil žrtvovati. Vseh

števil je 256 in če hočemo imeti nekaj negativnih števil, moramo na nekaj pozitivnih števil gledati kot na negativna. Smiselno je razdeliti na polovico. Števila od 0–127 ostanejo nedotaknjena (to so števila od 00000000 do 01111111) in jih interpretiramo kot prej. Števila od 128 do 255 pa interpretiramo kot negativna števila. Namreč 255 + 1 = 256, kar je v 8-bitnem sistemu enako kot 0. Podobno velja 254+2 = 0, 253+3 = 0 in na koncu 129 + 127 = 0. Odločimo se torej, da bomo $255 = 11111111_{(2)}$ gledali kot -1, in tako naprej do $129 = 10000001_{(2)}$ kot -127. Od tod tudi razlog za ime »komplement dvojice«: v n-bitnem sistemu število negiramo tako, da ga odštejemo od 2^n, v našem primeru od $2^8 = 256$. Rešiti moramo le še število 128, ki je samo sebi nasprotno število in ga lahko interpretiramo kot negativno ali pozitivno. Dogovor je, da ga gledamo kot negativno. To ima prednost, saj lahko predznačenost števila še vedno ugotovimo, tako da pogledamo prvi bit – 0 pomeni nenegativno, 1 pa negativno (le preostanek se drugače interpretira).

Naš razpon števil torej izgleda takole:

bitni zapis	nepredznačeno število	predznačeno število
00000000	0	0
00000001	1	1
00000010	2	2
⋮	⋮	⋮
01111110	126	126
01111111	127	127
10000000	128	−128
10000001	129	−127
10000010	130	−126
⋮	⋮	⋮
11111110	254	−2
11111111	255	−1

Zanimiva posledica takega zapisa je, kaj se zgodi, če imamo predznačeno celo 8-bitno število, ki je na začetku 0 in ga povečujemo za 1 v neskončni zanki. Vrednost bo rasla do $127 = 01111111_{(2)}$, nakar bo postala 10000000, kar si interpretiramo kot -128. Ko na desni strani prekoračimo največjo vrednost, pridemo ven pri najmanjši vrednosti. Če sedaj nadaljujemo, bo naslednja vrednost, ko prištejemo 1, enaka 10000001, kar si interpretiramo kot -127. Nato bo sledila 10000010, kar je -126. Vidimo, da se povečevanje pravilno obnaša, razen ko preskoči (kar ne drži pri predstavitvi, opisani na začetku). To pomeni, da lahko računalnik s števili pri seštevanju in odštevanju dela, kot da so pozitivna, in samo na koncu drugače pogleda na rezultat.

Naredimo en primer s polnimi 32-bitnimi števili. Najprej izračunajmo razpon predstavljivih števil v n-bitnem sistemu. V zadnjih $n-1$ bitov kot običajno zapišemo števila od 0 do $2^{n-1}-1$. Preostala števila od -2^{n-1} do -1 so negativna. V 32-bitnem sistemu je torej največje število $2^{31}-1 = 2147483647$, najmanjše pa $-2^{31} = -2147483648$.

Primer, ki si ga je enostavno zapomniti, je število z zapisom

- 11111111 11111111 11111111 11111111.

To število je -1, kar hitro vidimo na dva načina. Kot pozitivno število je enako $2^{32}-1$, torej je njegova vrednost $2^{32} - (2^{32}-1) = 1$, obravnavamo pa ga kot negativno, ker ima prvi bit 1. Še lažje, pogledamo, kaj se zgodi, če prištejemo 1: dobimo same 0 in enico, ki pade čez rob – torej je to število plus 1 enako 0 in je število torej -1.

Za vajo si sedaj interpretirajmo še število z zapisom

- 10100011 01011000 10110001 10101001.

Običajno interpretirano je to število 2 740 498 857. Toda vidimo, da ima na prvem mestu 1, in si ga moramo interpretirati kot negativno: izračunamo $2^{32} - 2\,740\,498\,857 = 1\,554\,468\,439$ in vemo, da zgornji zapis predstavlja -1554468439. S tem lahko tudi seštevamo, kot da delamo s pozitivnimi števili. Izračunajmo na ta način

- $-1554468439 + (-5)$,

torej $2\,740\,498\,857 + 4\,294\,967\,291 = 7\,035\,466\,148$, kar v bitih izgleda kot

- 10100011 01011000 10110001 10101001
+ 11111111 11111111 11111111 11111011
= 1 10100011 01011000 10110001 10100100

Enica, ki gleda čez, odpade in preostane vrednost, ki si jo interpretiramo kot -1554468444, kar je pravilen rezultat. Enako velja tudi za prištevanje pozitivnih števil. Izračunajmo npr. $-1554468439 + 1554468444$ in pričakujemo rezultat 5. Res, $2\,740\,498\,857 + 1\,554\,468\,444 = 4\,294\,967\,301 = 2^{32} + 5$, oz. v dvojiškem

- 10100011 01011000 10110001 10101001
+ 01011100 10100111 01001110 01011100
= 1 00000000 00000000 00000000 00000101,

kar je po tem, ko odrežemo enico, enako 5.

Za konec še trik, kako lahko hitro dobimo bitni zapis nasprotnega števila, če že poznamo bitni zapis originala. Postopek gre takole: vzamemo bitni zapis originala in obrnemo vse bite (0 postane 1 in obratno). Nato prištejemo 1.

Najprej preizkusimo na primeru, potem pa razložimo, zakaj deluje. Vzemimo število 19 z zapisom

- 00000000 00000000 00000000 00010011.

Obrnemo bite in dobimo

- 11111111 11111111 11111111 11101100.

Prištejemo 1, dobimo

- 11111111 11111111 11111111 11101101

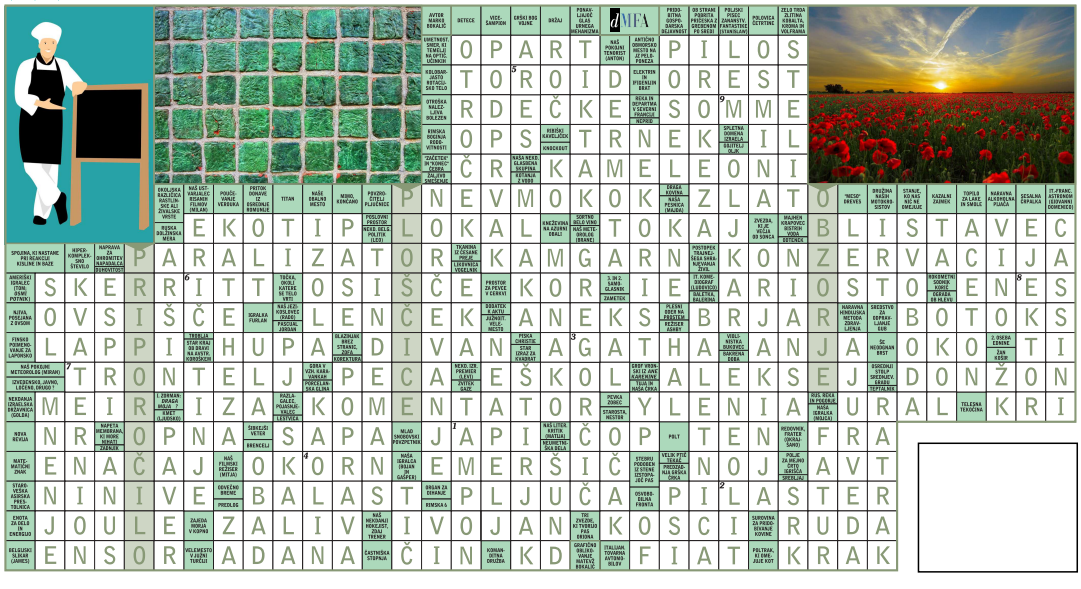
in trdimo, da je to -19. Pa poglejmo: kot pozitivno število je to enako $4\,294\,967\,277$, toda ko ga seštejemo z 19 dobimo točno 2^{32}, kar je v 32-bitnih številih enako kot 0, torej je moralo število res biti -19.

Premislimo še, zakaj postopek res deluje. Imejmo na začetku število X, obrat bitov pa označimo z $\sim X$. Pokazati želimo, da je $\sim X + 1$ nasprotno število od X. To je najlažje storiti tako, da ju seštejemo, in moramo dobiti 0. Poglejmo, kaj se zgodi, ko seštevamo $\sim X$ in X. Ker so vsi biti nasprotni, je rezultat vedno enak $11111\ldots11$. Ko temu prištejemo 1, dobimo ravno same ničle in enico, ki pade čez rob, torej je rezultat res 0.

Literatura

[1] D. Cohen, *On holy wars and a plea for peace*, Computer, 14(10), 48–54, 1981, dostopno na `www.ietf.org/rfc/ien/ien137.txt`, ogled: 5. 8. 2020.

× × ×



REŠITEV NAGRADNE KRIŽANKE

PRESEK 47/6

→ Pravilna rešitev nagradne križanke iz šeste številke Preseka je **Algoritem**. Izmed pravilnih rešitev so bili izžrebani Vid Kavčič iz Črnomlja, Martin Mah iz Šmartnega pri Litiji in Ema Hajnšek iz Ljubljane, ki bodo razpisane nagrade prejeli po pošti.

× × ×