Rank a select

V tejto kapitole si vybudujeme jeden z najzákladnejších stavebných kameňov pre úsporné dátové štruktúry: *bitvektor* s operáciami `access`, `rank` a `select`.

Majme bitvektor $B \in \mathtt{0},\mathtt{1}^n$. Na ňom budeme definovať tri základné operácie:

- access(i) (prístup) vráti bit $B[i]$.
- $\mathrm{rank}_{\mathtt{1}}(i)$ vráti počet jednotiek v prefixe $B[0 \ldots i)$ pred pozíciou i. Analogicky $\mathrm{rank}_{\mathtt{0}}(i) = i - \mathrm{rank}_{\mathtt{1}}(i)$ udáva počet núl.
- $\mathrm{select}_{\mathtt{1}}(k)$ vráti najmenšie i, pre ktoré $\mathrm{rank}_{\mathtt{1}}(i+1) = k$, t. j. pozíciu k-tej jednotky. Analogicky definujeme $\mathrm{select}_{\mathtt{0}}$ pre nuly.[1]

Na prvý pohľad to nevyzerá ako nič zvláštne – prístup do poľa, počet jednotiek, pozícia jednotky. Lenže cieľom bude navrhnúť tieto operácie tak, aby boli čo najrýchlejšie (v konštantnom čase) a zároveň zaberali *čo najmenej pamäte*.

Motivácia #1: FM-index. V kapitole o FM-indexe sme narazili na potrebu rýchlo spočítať, koľkokrát sa daný znak `c` vyskytuje v prefixe $L[0 \ldots i)$. Pre binárnu abecedu je táto úloha presne operácia rank nad bitvektorom. Keď tento prípad zvládneme rýchlo a pamäťovo úsporne, v ďalšej kapitole sa pozrieme na väčšie abecedy.

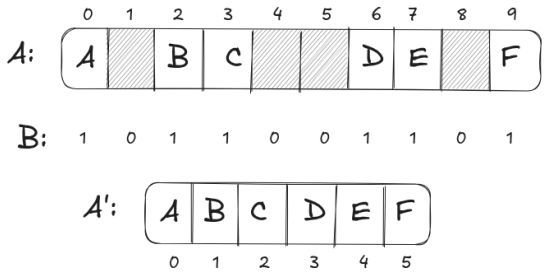
Motivácia #2: Zhustené pole. Predstavme si pole A dĺžky m, ktoré obsahuje len $n < m$ reálnych prvkov a zvyšok je prázdny. Každý prvok zaberá ℓ bitov a predpokladajme, že v týchto ℓ bitoch vieme reprezentovať aj špeciálnu hodnotu „prázdny“ (napríklad `null`). Ako môžeme takéto riedke pole reprezentovať úsporne a pritom efektívne?

Jednoduché riešenie: obyčajné pole. Najpriamejší prístup je alokovať $m \times \ell$ bitov, teda klasické pole dĺžky m. Nevýhoda je zrejmá – ak je pole riedke, väčšina priestoru sa spotrebuje na prázdne políčka. Na druhej strane, ak je n len o trochu menšie ako m, teda prázdnych miest je málo, je to dobré riešenie.

Druhá možnosť je uložiť si iba tie prvky, ktoré skutočne existujú, ako slovník $i \mapsto A_i$. Takýto slovník môžeme implementovať buď ako utriedené pole (v ktorom budeme binárne vyhľadávať), alebo ako hešovaciu tabuľku. Takéto riešenie zaberá približne $n \times (\lg m + \ell)$ bitov, v prípade heštabuľky ešte krát $(1 + \varepsilon)$. Toto je výborné riešenie, ak je pole veľmi riedke, teda $n \ll m$.

Keď si o chvíľu ukážeme úspornú implementáciu pre operácie `rank` a `select`, pribudne nám ešte tretia možnosť: Vytvoríme si bitvektor B dĺžky m, v ktorom si pre každú pozíciu uložíme informáciu, či je dané políčko obsadené (`1`) alebo prázdne (`0`). Zároveň si alokujeme pole A' dĺžky n, do ktorého zapíšeme všetky neprázdne prvky A_i, natlačené tesne vedľa seba, bez medzier.

[1] Pozor na off-by-one chyby: v celej knihe budeme dôsledne používať nulové indexovanie a intervaly tvaru $[0 \ldots i)$, takže rank(i) sa vždy počíta len po pozíciu i, nie vrátane nej



Ak potom chceme zistiť, či je $A[i]$ obsadené, jednoducho sa pozrieme na bit $B[i]$. Ak $B[i] = \texttt{1}$, vieme pomocou operácie $\mathrm{rank}_1(i)$ zistiť, koľko neprázdnych prvkov sa nachádza pred pozíciou i, a teda aj index v poli A', kde sa nachádza požadovaná hodnota:

$$A[i] = A'[\mathrm{rank}_1(i)].$$

Naopak, operácia select_1 nám umožňuje nájsť, kde sa v pôvodnom poli nachádza j-ty neprázdny prvok, teda slúži na prepočet indexov opačným smerom:

$$A'[j] = A[\mathrm{select}_1(j)].$$

Takéto riešenie zaberá $(1 + \varepsilon)m + n \times \ell$ bitov.

Ktoré riešenie je najlepšie? Záleží...

Predpokladajme, že implementácia `rank` a `select`, resp. heštabuľky zaberie $\varepsilon = 10\%$ pamäte navyše. Vezmime napríklad $n = 2^{20} \approx 1,000,000$ prvkov, pričom každý z nich má veľkosť $\ell = 256$ bitov (teda 32 bajtov). Samotné dáta teda zaberajú $n \times \ell = 2^{20} \times 256 = 32\,\mathrm{MB}$. Porovnajme teraz, koľko pamäte zaberie celá štruktúra pri rôznych hustotách:

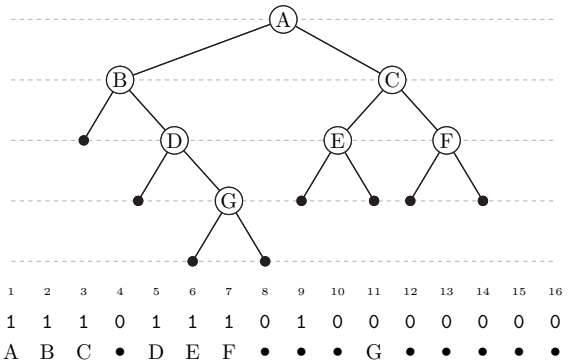
	riedke pole	heštabuľka	zhustené pole s bitvektorom
pamäť	$m \times \ell$	$(1 + \varepsilon) \times n \times (\lg m + \ell)$	$(1 + \varepsilon)m + n \times \ell$
$m = 1.1n$	35.2 MB	37.97 MB	32.15 MB
$m = 2n$	64 MB	38.1 MB	32.28 MB
$m = 10n$	320 MB	38.4 MB	33.38 MB
$m = 50n$	1.56 GB	38.73 MB	38.88 MB
$m = 100n$	3.13 GB	38.86 MB	45.75 MB

Inými slovami, *dodatočná pamäť* slovníka $i \mapsto A_i$ je prinajmenšom $n \lg m$ bitov navyše – toľko potrebujeme len na uloženie indexov. V prípade zhusteného poľa s bitvektorom je dodatočná pamäť približne $(1 + \varepsilon) \times m$ bitov, teda len o málo viac než samotná dĺžka pôvodného poľa. Z toho vyplýva, že použitie slovníka sa oplatí až vtedy, keď $m \gg n \lg m$, teda keď je pole skutočne veľmi riedke.

Motivácia #3: Úsporné stromy. Predstavme si, že máme statický binárny strom a chceme ho uložiť čo najúspornejšie. Tradičná reprezentácia – teda každému vrcholu uložiť smerníky na ľavého a pravého syna a rodiča – je síce pohodlná, ale z pohľadu pamäte veľmi neefektívna. Každý smerník zaberá 64 bitov, takže len samotné odkazy pridávajú až $3 \times 64 = 192$ bitov na každý jeden vrchol.

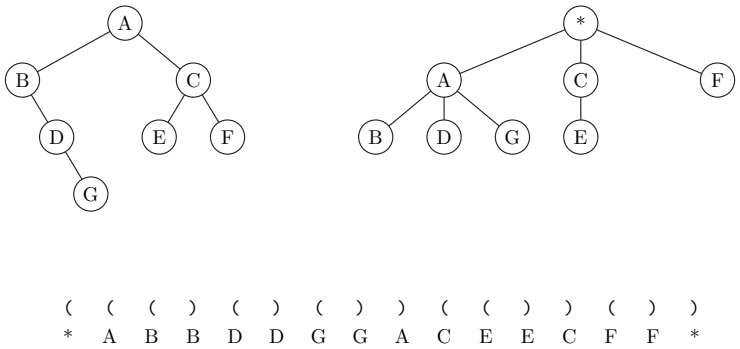
Dá sa to lepšie?

Áno: Doplňme strom o tzv. *vonkajšie vrcholy* (•), ktorými nahradíme všetky nulové smerníky. Každý vnútorný vrchol má teda presne dvoch synov – buď vnútorných, alebo vonkajších. Teraz stačí prejsť strom po úrovniach a zapísať si `1` ak je daný vrchol vnútorný, `0` ak je vonkajší:



Týmto spôsobom dokážeme binárny strom reprezentovať ako obyčajný bitvektor, ktorý má dĺžku $2n + 1$ bitov. Navyše, vďaka operáciám `rank` a `select` nad týmto bitvektorom sa dokážeme po strome pohybovať – z rodiča na deti, z detí na rodiča. (Ako presne, necháme ako úlohu pre čitateľa.)

Alternatívny prístup je previesť binárny strom na *zakorenený usporiadaný strom* pomocou tzv. reprezentácie *left-child–right-sibling*: V tejto schéme interpretujeme ľavého syna ako *prvého potomka* a pravého syna ako *nasledujúceho súrodenca*.



(Napríklad v pôvodnom binárnom strome koreň A, jeho pravý syn C a *jeho*

pravý syn F tvoria súrodencov – synov nového koreňa označeného $(*)$. Ľavý syn B sa v tejto reprezentácii stáva *prvým* synom vrcholu A, a jeho praví potomkovia sa premenia na jeho súrodencov.)

Nakoniec spravíme Eulerovský prechod okolo nového stromu. Vždy, keď do vrcholu prvýkrát vchádzame, zapíšeme si ľavú zátvorku, `(` a keď z neho naposledy odchádzame, zapíšeme pravú zátvorku `)`. Takto dostaneme *dobre uzátvorkovanú postupnosť*, ktorá presne vystihuje tvar pôvodného stromu.

Tieto dve transformácie – *left-child–right-sibling* a *Eulerovský prechod* – sú izomorfizmy: každému binárnemu stromu zodpovedá práve jeden zakorenený usporiadaný strom, a každej takejto štruktúre prislúcha práve jedna dobre uzátvorkovaná postupnosť. Naopak, z každej správne uzátvorkovanej postupnosti vieme strom jednoznačne zrekonštruovať. Zároveň postupnosť zátvoriek môžeme reprezentovať ako bitový vektor, kde `(` je `0` a `)` je `1`. Opäť prenecháme na čitateľa, aby domyslel, ako operácie na stromoch premeniť na operácie na bitvektore.

Vidíme teda, že bitvektory a operácie na nich nie sú len akýsi podivný umelý problém, ktorým sa teoretický informatici zaoberajú počas dlhých zimných večerov, ale tvoria praktický nástroj, na ktorom je založených veľa ďalších úsporných dátových štruktúr. Poďme teraz porozmýšľať, ako takúto dátovú štruktúru navrhnúť.

Úsporný rank

Začnime dvoma triviálnymi riešeniami:

#0. Žiadne predspracovanie: Rank spočítame v lineárnom čase. Katastrofa, ale OK, musel som to spomenúť.

#1. Predpočítame všetko: Spočítame si tabuľku $R[i] = \mathrm{rank}_1(i)$ – každý rank takto vieme povedať v konštantnom čase. Avšak pamäť je katastrofálna: napríklad ak použijeme 64-bitové čísla, tak pole rankov bude 64×väčšie ako pôvodný bitvektor!

Vieme vymyslieť niečo lepšie?

#2. Predpočítame menej: Ak chceme zmenšiť potrebnú pamäť, čo tak predpočítať si rank len pre každú k-tu pozíciu? Každý rank zaberá $\lceil \lg n \rceil$ bitov, ale ak si zapamätáme iba každé k-te, celkovo ranky zaberú $(n/k) \times \lg n$ bitov.

Napríklad, ak zvolíme $k = 10 \lg n$, ranky spotrebujú len 10% pamäte navyše oproti samotnému bitvektoru.

Lenže na druhej strane, ak chceme, aby ranky zaberali menej ako n bitov, *potrebujeme* $k > \lg n$, ale tým pádom sa nám zhorší časová zložitosť. Ak si pamätáme len každý $(\log n)$-tý rank, tie zvyšné musíme dopočítať v lineárnom čase od k. Či?

#3. Praktické riešenie: Poďme sa najskôr porozprávať o reálnej implementácii na reálnom počítači. Pretože $k = 10 \lg n$ znie hrozivo, ale ak máme povedzme miliardové pole, tak $\lg 10^9 < 30$, takže máme jeden rank pre každých 300 bitov. Pre predstavu, jeden register je 64 bitov a jedna cache line je 64B = 512 bitov.

Navyše súčasné počítače majú inštrukciu `popcount`, ktorá spočíta počet jednotkových bitov v registri. Jediná inštrukcia v konštantnom čase. Takže 300 bitov zvládneme 5-timi inštrukciami, ktoré môžu bežať dokonca paralelne. Dokonca najnovšie počítače (v čase písania) s architektúrou AVX-512 (napríklad Intel Ice Lake) majú vektorové inštrukcie `VPOPCNTDQ`, ktoré spočítajú počet jednotiek v 8-mich 64-bitových číslach *naraz*, dokonca s prípadnou maskou.

Ak chceme spočítať len počet jednotiek po nejakú pozíciu i, stačí bity od tejto pozície vynulovať pomocou bitového `AND`u. Konkrétne stačí zobrať `1`, posunúť ju o i doľava a odčítať 1, teda `(1 << i) - 1`. Dostaneme v dvojkovej sústave:

$$\begin{array}{rl} \texttt{1 << i} & \texttt{00001}\overbrace{\texttt{000 ... 000}}^{i} \\ \texttt{(1 << i) - 1} & \texttt{00000111 ... 111} \end{array}$$

Keď ľubovoľné číslo z`AND`ujeme s takouto maskou, ostanú nám len bity na jednotkových pozíciach, teda prvých i pozícií (sprava).

Jedno veľmi praktické riešenie teda je, že vezmeme 512 bitov (64 bajtov, 1 cache line). Z toho prvých 64 bitov bude predpočítaný rank, teda počet jednotiek pred touto pozíciou a zvyšných 448 bitov budú bity pôvodného bitvektoru. Dôležité nemáme dve oddelené polia: pôvodný bitvektor a predpočítané ranky, ale všetky hodnoty sú uložené prerývane v jednom poli. Takto nám stačí namiesto dvoch prístupov na dve rôzne miesta v pamäti načítať naraz len jednu cache line. Výsledný rank spočítame ako súčet predpočítaného ranku a hodnoty dopočítanej maximálne 7-mimi operáciami `popcount`, s prípadným maskovaním. Takéto riešenie zaberá $\approx 14\%$ pamäte navyše oproti samotnému bitvektoru. Dokonca ak máme bitvektor dĺžky menej ako 4 miliardy, stačí nám len 32 bitov na každý rank a zvyšných 480 bitov tvoria bity z bitvektoru, čo je len $1/15 \approx 6.7\%$ navyše.

Čas je prakticky konštantný. (Teda ak predpokladáme model počítača, v ktorom máme registre dĺžky $\Theta(\log n)$ a `popcount` vieme spočítať v konštantnom čase.)

Dá sa to ešte lepšie? Čo keby sme chceli ešte úspornejšie riešenie?

Predstavme si, že máme $n = 10\,\text{GB}$ dlhý bitvektor. To znamená, že na ranky potrebujeme aspoň 34 bitov ($\lceil \lg(10 \cdot 2^{30}) \rceil$). Ak bitvektor nasekáme na bloky dĺžky $b = 512$, potrebujeme aspoň

$$\frac{n}{b} \times 34 = n \times 34/512 = 680\,\text{MB} \approx 6.6\% \times n$$

Vieme sa dostať pod 6%? Mohli by sme ešte predĺžiť veľkosť bloku, lenže tým zároveň algortimus spomaľujeme...

#4. Ešte úspornejšie riešenie: Môžeme však použiť dvoj-úrovňové riešenie! Nasekáme bitvektor na superbloky dĺžky $s = 2^{14} - 1 = 16383$ bitov. Pre každý

začiatok superbloku spočítame rank po danú pozíciu. Ostáva nám vyriešiť, ako spočítame rank v rámci jedného superbloku.

Odpoveď je, že môžeme každý superblok opäť nasekať na menšie bloky dĺžky $b = 512$ (a v rámci bloku dĺžky 512 už rank spočítame pomocou bitových operácií). Vtip je však v tom, že pre tieto bloky stačí predpočítať rank iba *od začiatku superbloku*, čo je číslo od 0 po 16383, na ktoré nám stačí 14 bitov!

Koľko pamäte takéto riešenie spotrebuje? Máme n/s superblokov a na každý rank potrebujeme 34 bitov, plus máme n/b blokov a na každý blok predpočítame „malý rank“ od začiatku superbloku, na ktorý potrebujeme len 14 bitov. Spolu:

$$\frac{n}{s} \times 34 + \frac{n}{b} \times 14 = n \times \left(\frac{34}{2^{14} - 1} + \frac{14}{512} \right) \approx 301\,\text{MB} \approx 2.94\% \times n$$

Ak by sme zvolili bloky dĺžky $b = 1024$, oplatí sa zvoliť superbloky dĺžky $s = 2^{15} - 1 = 32767$. Pri tejto voľbe by ranky zaberali

$$n \times \left(\frac{34}{2^{15} - 1} + \frac{15}{1024} \right) \approx 161\,\text{MB} \approx 1.57\% \times n$$

Teória. V teórii nás zaujímajú *úsporné* riešenia, kde dodatočná pamäť je *asymptoticky menšia* ako minimálna pamäť pre danú štruktúru – v našom prípade $o(n)$. To znamená, že chceme dosiahnuť takú pamäť, že podiel

$$\frac{\text{dodatočná pamäť na ranky}}{\text{pamäť na bitvektor}} \to 0 \qquad \text{pre } n \to \infty$$

je zanedbateľný, resp. blíži sa k nule pre n idúce do nekonečna.

Vo všeobecnosti v dvojúrovňovom riešení ranky zaberajú

$$n \times \left(\frac{\lceil \lg n \rceil}{s} + \frac{\lceil \lg s \rceil}{b} \right) \text{ bitov.}$$

Ak zvolíme napríklad superbloky dĺžky $s = \lg^2 n$ a bloky dĺžky $b = \frac{1}{2} \lg n$, tak ranky budú zaberať

$$n \times \left(\frac{\lceil \cancel{\lg n} \rceil}{\lg n \cdot \cancel{\lg n}} + \frac{\lceil \lg(\lg^2 n) \rceil}{\frac{1}{2} \lg n} \right) = n \times O\Bigg(\underbrace{\frac{1 + \lg \lg n}{\lg n}}_{\to 0 \text{ pre } n \to \infty} \Bigg) = o(n)$$

A čo sa týka počítania ranku v rámci jedného bloku (dĺžky $\frac{1}{2} \lg n$), tak aj keby sme neuvažovali model s operáciou `popcount`, tak pri takejto malej dĺžke si vieme predpočítať jednu globálnu tabuľku. Všetkých možných bitvektorov dĺžky b je totiž 2^b, čo pre našu voľbu $b = \frac{1}{2} \lg n$ znamená $\sqrt{n}$. Aj keby sme si teda pre každý bitvektor a pre každú pozíciu v ňom spočítali rank, zaberie to iba $O(\sqrt{n} \times \lg^2 n)$ bitov, čo je zanedbateľné oproti n.

Úsporný select

Operácia $\text{select}_1(j)$ je v istom zmysle inverzná ku $\text{rank}_1(i)$. Pri jeho riešení budeme vyzbrojení technikami, ktoré sme použili na rank, napriek tomu je tento problém zložitejší. Pri ranku nám stačilo vyriešiť rank_1 a počet núl sme mohli počítať vďaka vzťahu $\text{rank}_0(i) = i - \text{rank}_1(i)$. Naproti tomu ak vieme nájsť j-tu jednotku, to nám ešte nič nehovorí o tom, kde sa nachádza j-ta nula. Naše úsilie budeme musieť zdvojnásobiť a vyrobiť jednu štruktúru pre select_0 a jednu pre select_1.

Obvyklé triviálne riešenia stále fungujú (a stále sú katastrofálne zlé):

#0. Žiadne predspracovanie: Hľadanie trvá $O(n)$.

#1. Predpočítame všetko: Zaberá $O(n \log n)$ bitov pamäte navyše. A my chceme $o(n)$.

V skutočnosti, ak nás zaujíma iba select_1, je to $O(n_1 \log n)$ bitov, kde n_1 je počet jednotiek v poli. Táto reprezentácia sa však oplatí iba ak je počet jednotiek dosť malý – $n_1 = o(n/\log n)$.

#2. Binárne vyhľadávanie na rankoch: Jedno veľmi praktické riešenie, ak už máme vybudovanú štruktúru pre ranky, je použiť binárne vyhľadávanie a najskôr medzi rankami superblokov nájsť ten správny superblok, potom medzi rankami blokov nájsť ten správny blok a nakoniec v rámci bloku dohľadať tú správnu jednotku. Časová zložitosť bude $O(\log n)$, čo nie je na zahodenie a výhodou je, že okrem štruktúry pre rank už nepotrebujeme žiadne ďalšiu pamäť navyše.

#3. Predpočítame menej: Keďže predpočítať všetko má príšernú pamäťovú zložitosť, ale predpočítanie vo všeobecnosti pomáha, môžeme ušetriť pamäť tak, že si toho predpočítame menej. Ak si zapamätáme iba pozíciu každej k-tej jednotky, pamäťová zložitosť bude $O(n/k \times \log n)$, čo, povedzme, pre $k = \log^2 n$, je sublineárne $o(n)$.

Človek by čakal, že takto zlepšíme časovú zložitosť, ale v najhoršom prípade tomu tak nie je. Človek by tiež čakal, že takto nasekáme bitvektor na menšie bloky, použijeme dvoj-úrovňové riešenie ako pri rankoch a môžeme si odfajknúť select a rozlúčiť sa. Bohužiaľ, nie je to také ľahké.

Totiž zatiaľčo pri rankoch sme bitvektor nasekali na rovnaké bloky dĺžky b, pri selecte budú mať bloky rôzne dĺžky. Dokonca veľmi dramaticky rôzne: napríklad ak vieme, že $\text{select}_1(0) = 0$, tak $\text{select}_1(1)$ môže byť hneď vedľa na pozícii 1, alebo aj úplne na konci $\text{select}_1(1) = n - 1$, ak sú medzitým samé nuly.

Našťastie nás zachráni, že pre riedke polia, kde je veľmi málo jednotiek, môžeme použiť aj triviálne riešenie – a síce poznačíme si pozície všetkých jednotiek.

#4. Plus všetky pozície v dlhých blokoch: Predpočítajme si pozíciu každej k-tej jednotky, kde $k = \log^2 n$. Tieto pozície nám rozdelia bitvektor na men-

šie bloky. Bloky môžu mať veľmi rôznorodé dĺžky, ale v každom bloku (možno okrem posledného) sa nachádza presne k jednotiek.

Ak je blok dlhší ako $k^2 = \log^4 n$, budeme ho volať *dlhý* blok – ostatné budeme volať *krátke*. Zjavne dlhých blokov môže byť najviac n/k^2 a v každom je najviac k jednotiek – to je dokopy najviac $n/k = n/\log^2 n$ jednotiek. Ak si aj pre každú z nich zapamätáme jej pozíciu, zaberie to dokopy len $O(n/\log n) = o(n)$ pamäte. Navyše pre každý z n/k blokov si potrebujeme poznačiť, či je dlhý alebo krátky a pre dlhé bloky smerník na pole všetkých pozícii – na toto opäť môžeme použiť bitvektor s predpočítaným rankom, ale aj priamočiarejšie riešenia budú zaberať len $o(n)$ pamäte.

Keď takto vyriešime všetky dlhé bloky, ostáva nám doriešiť tie krátke. Každý z nich má dĺžku najviac $\log^4 n$, takže ak použijeme binárne vyhľadávanie, dostaneme riešenie s časovou zložitosťou $O(\log(\log^4 n)) = O(\log \log n)$.

#5. Dvoj-úrovňové riešenie. Budeme postupovať rovnako ako v predchádzajúcom riešení – predpočítame si pozíciu každej k-tej jednotky pre $k = \log^2 n$. Pre dlhé bloky si predpočítame pozície všetkých jednotiek. Pre krátke bloky použijeme tú istú myšlienku ešte raz.

Vyderžaj pioner.

Pre každý *krátky* blok si predpočítame každú ℓ-tú jednotku pre $\ell = (\log \log n)^2$. Vtip je opäť v tom, že pozíciu budeme počítať od začiatku bloku a keďže všetky krátke bloky majú dĺžku najviac $\log^4 n$, táto pozícia zaberá len $4 \log \log n$ bitov. Spolu všetky tieto pozície zaberú najviac

$$\frac{n}{\ell} \times 4 \log \log n = O\left(\frac{n \cancel{\log \log n}}{\log \log n \times \cancel{\log \log n}}\right) = o(n) \text{ bitov.}$$

Tieto pozície nám rozdelia každý krátky blok na menšie *minibloky*. Minibloky budú mať opäť rôznu dĺžku. Budeme hovoriť že minibloky dlhšie ako $\ell^2 = (\log \log n)^4$ sú *dlhé* a tie ostatné sú krátke. Pre dlhé minibloky si môžeme opäť uložiť všetky pozície jednotiek – takýchto dlhých miniblokov je len n/ℓ^2 a každý obsahuje najviac ℓ jednotiek – to je spolu $n/\ell = n/(\log \log n)^2$ jednotiek. Každá pozícia zaberá len $\log \log n$ bitov, takže si to môžeme dovoliť. Celková pamäť pre dlhé minibloky je

$$O\left(\frac{n}{\ell^{\cancel{2}}} \times \cancel{\ell} \times \log \log n\right) = O\left(\frac{n}{(\log \log n)^{\cancel{2}}} \times \cancel{\log \log n}\right) = O\left(\frac{n}{\log \log n}\right) = o(n).$$

Nakoniec nám ostanú krátke minibloky dĺžky najviac $\ell^2 = (\log \log n)^4$ a tie sa už zmestia do registra, resp. všetkých možných bitvektorov takejto dĺžky je len veľmi málo, takže si vieme predpočítať jednu globálnu tabuľku, ktorá nám povie pre každý bitvektor a každé j pozíciu j-tej jednotky.

Takto dokážeme rank aj select spočítať v konštantnom čase a v $o(n)$ dodatočnej pamäti.

Komprimovaný bitvektor

Ako sme videli v predchádzajúcej časti, ak máme reťazec bitov, vieme si k nemu uložiť pomocné informácie a vďaka nim podporovať operácie rank a select. Celá táto sranda je veľmi praktická a stojí nás len malú pamäť, rádovo jednotky percent z pamäte, ktorú zaberá pôvodný reťazec.

V nasledujúcej časti skúsime byť ešte ambicióznejší. Vedeli by sme daný bitvektor *skomprimovať* tak, že stále budeme vedieť podporovať operácie access, rank a select? A za akú cenu?

Samozrejme, mohli by sme zo šuflíka vytiahnuť ľubovoľný kompresný algoritmus, a celý bitvektor skomprimovať – ale ako potom budeme vedieť zistiť hodnotu i-teho bitu bez toho, aby sme celý reťazec dekomprimovali?

Ďalší nápad by mohol byť rozsekať vstupný bitvektor na bloky (nápad „sekať na bloky“ nás už viackrát zachránil). Každý blok zakódujeme zvlášť. Budeme si musieť zapamätať nejakú pomocnú infomáciu, aby sme rýchlo vedeli nájsť k-ty blok, avšak potom stačí na zistenie i-teho bitu dekomprimovať iba jediný blok a nie celý reťazec. Navyše, myšlienka blokov ide pekne dokopy s algoritmami pre rank a select, ktoré tiež delia bitvektor na bloky.

Stojíme však pred neľahkou úlohou. Na jednej strane chceme bloky čo najdlhšie, aby sme mali čo najlepšiu kompresiu a potrebovali čo najmenej dodatočnej pamäte pre rank a select, na druhej strane chceme bloky krátke, aby sme vedeli rýchlo dekódovať i-ty bit.

Tu si ukážeme jednu jednoduchú metódu, ako bitový reťazec skomprimovať, pričom sa budeme blížiť entropii daného reťazca. Dátová štruktúra sa volá RRR podľa mien jej vynálezcov: Raman, Raman a Rao.

Myšlienka je jednoduchá:

- Vezmime blok r bitov.
- Spočítajme c, počet jednotiek v ňom.
- Predstavme si, že máme utriedený zoznam všetkých bitvektorov dĺžky r, ktoré obsahujú c jednotiek a spočítajme poradie p nášho bloku v tomto zozname.
- Náš blok budeme kódovať dvojicou (c, p).

Ukážme si to na príklade. Predstavme si, že máme na vstupe:

0110000	0001000	0110010	0000000	0000010	0000101

a zvolíme si blok dĺžky 7. Pomocná tabuľka je na obrázku 1. Prvý blok má 2 jednotky a medzi takými blokmi je 14-ty v poradí (počítame od nuly). Druhý blok má 1 jednotku a je 3-tí v poradí. Tretí blok má 3 jednotky a je to 17-ty blok medzi 7-bitovými blokmi s 3-oma jednotkami. Štvrtý blok sú samé nuly, teda $c = 0$ a žiadne poradie nepotrebujeme kódovať, pretože existuje len jeden taký blok, atď. Výsledný kód bude:

c	2	1	3	0	1	2
p	01110	011	010001		001	00001

Žiadna jednotka

0	0000000	→	ε

Jedna jednotka

0	0000001	→	000
1	0000010	→	001
2	0000100	→	010
3	0001000	→	011
4	0010000	→	100
5	0100000	→	101
6	1000000	→	110

Dve jednotky

0	0000011	→	00000
1	0000101	→	00001
2	0000110	→	00010
3	0001001	→	00011
	...		...
14	0110000	→	01110
15	1000001	→	01111
16	1000010	→	10000
	...		...
20	1100000	→	10100

Tri jednotky

0	0000111	→	000000
1	0001011	→	000001
2	0001101	→	000010
	...		...
8	0011010	→	001000
9	0011100	→	001001
10	0100011	→	001010
11	0100101	→	001011
12	0100110	→	001100
13	0101001	→	001101
14	0101010	→	001110
15	0101100	→	001111
16	0110001	→	010000
17	0110010	→	010001
18	0110100	→	010010
19	0111000	→	010011
20	1000011	→	010100
	...		...
31	1100010	→	011111
32	1100100	→	100000
33	1101000	→	100001
34	1110000	→	100010

Obr. 1: Všetky 7-bitové reťazce môžeme rozdeliť do skupín podľa počtu jednotiek (na obrázku sú len skupiny s 0–3 jednotkami). V rámci každej skupiny si vypíšeme reťazce s daným počtom jednotiek a očíslujeme. Keďže existuje 7 reťazcov s 1 jednotkou, na zapísanie poradia v tejto skupine nám stačia 3 bity. Reťazcov s 2-oma jednotkami je $\binom{7}{2} = 21$, čo vieme zapísať 5 bitmi a reťazcov s 3-omi jednotkami je $\binom{7}{3} = 35$, na čo potrebujeme 6 bitov. Tabuľky pre 4–7 jednotiek budú vyzerať podobne, symetricky – stačí bity znegovať a očíslovať opačne.

Na zápis každého c potrebujeme 3 bity (môže nadobúdať $0 \ldots 7$, teda 2^3 rôznych hodnôt), takže dokopy táto reprezentácia zaberie $6 \times 3 + 5 + 3 + 6 + 3 + 5 = 40$ bitov, zatiaľčo pôvodná postupnosť mala $6 \times 7 = 42$ bitov.

To nie je žiadna sláva, lenže na lepšiu kompresiu treba zväčšiť dĺžku bloku r.

Vo všeobecnosti, ak máme bloky dĺžky r, tak c nadobúda hodnoty medzi 0 a r, čo je $r + 1$ možných rôznych hodnôt. Aby sme tieto hodnoty zapísali, potrebujeme $\lceil \lg(r+1) \rceil$ bitov. Je teda výhodné sústrediť sa práve na bloky dĺžky o 1 menej ako nejaká mocnina dvojky (v opačnom prípade plytváme bitmi).

Keď potom narazíme na blok, ktorý má c jednotiek, tak existuje presne $\binom{r}{c}$ bitvektorov s c jednotkami – jednoduchá kombinatorika: z r miest potrebujeme *vybrať podmnožinu* c pozícií, na ktorých sú jednotky, takže odpoveď je kombinačné číslo $\binom{r}{c}$, počet výberov c pozícií z r. Na reprezentáciu X rôznych možností potrebujeme $\lceil \lg X \rceil$ bitov, takže na poradie p budeme potrebovať práve $\lceil \lg \binom{r}{c} \rceil$ bitov.

Pre lepšiu predstavu si spočítajme, koľko to je konkrétne, pre rôzne dĺžky blokov.

Pre $r = 15$ sa má situácia nasledovne. Potrebujeme 4 bity na zápis c a počet bitov na zápis pozície je $\lceil \lg \binom{15}{c} \rceil$:

$r = 15$, c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\lceil \lg \binom{15}{c} \rceil$	0	4	7	9	11	12	13	13	13	13	12	11	9	7	4	0
$4 + \left\lceil \lg \binom{15}{c} \right\rceil$	4	8	11	13	15	16	17	17	17	17	16	15	13	11	8	4

Z tabuľky vidíme, že na blokoch, ktoré majú menej ako 4 jednotky, alebo viac ako 11 jednotiek, budeme šetriť pamäť. Bloky so 4 alebo 11 jednotkami budú zaberať rovnako 15 bitov v pôvodnom aj komprimovanom reťazci. Bloky, ktoré majú 5–10 jednotiek, sa nám ešte predĺžia o 1 alebo 2 bity. (Všimnite si, že tabuľka je symetrická podľa stredu, keďže kombinačné čísla sú symetrické – v tabuľke máme vlastne len zlogaritmovaný riadok Pascalovho trojuholníka.)

Ak zvolíme bloky dĺžky 31, budeme potrebovať 5 bitov na zápis c a na zápis pozície p budeme potrebovať:

$r = 31$, c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\lceil \lg \binom{31}{c} \rceil$	0	5	9	13	15	18	20	22	23	25	26	27	28	28	28	29
$5 + \left\lceil \lg \binom{31}{c} \right\rceil$	5	10	14	18	20	23	25	27	28	30	31	32	33	33	33	34

druhá polovica tabuľky je symetrická. Takže bloky s menej ako 10 alebo viac ako 21 jednotiek z 31 sa skracujú a bloky s 11–20-timi jednotkami sa mierne predlžujú.

Nakoniec ak zvolíme bloky dĺžky 63, na zápis c potrebujeme 6 bitov a na zápis pozície:

$r = 63$, c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\lceil \lg \binom{63}{c} \rceil$	0	6	11	16	20	23	27	30	32	35	37	40	42	44	46	47
$6 + \left\lceil \lg \binom{63}{c} \right\rceil$	6	12	17	22	26	29	33	36	38	41	43	46	48	50	52	53

c	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$\lceil \lg \binom{63}{c} \rceil$	49	50	52	53	54	55	56	57	58	58	59	59	60	60	60	60
$6 + \left\lceil \lg \binom{63}{c} \right\rceil$	55	56	58	59	60	61	62	63	64	64	65	65	66	66	66	66

Vidíme, že bloky s menej ako 23 a viac ako 40 jednotkami sa skracujú a bloky s 24–39 jednotkami sa predlžujú.

Aká dobré je takáto kompresia?

Predstavme si, že vygenerujeme úplne náhodný reťazec, ktorý má len 5% jednotiek (každý jeden bit vygenerujeme úplne nezávisle tak, že si hodíme mincou, na ktorej padá jednotka s 5% pravdepodobnosťou). Shannonova entropia takéhoto zdroja je $H(5\%) \approx 0.286$, teoreticky sa takýto reťazec nedá skomprimovať lepšie ako na $\approx 29\%$. Ak zvolíme RRR kódovanie a bloky dĺžky 63,

vieme skrátiť (očakávane) reťazec na $\approx 33.7\%$. Pre rôzne počty jednotiek a rôzne veľkosti blokov:

		očakávaná kompresia s RRR		
#jednotiek	*entropia*	$r = 127$	$r = 63$	$r = 31$
5%	0.286	31.5%	33.7%	37.6%
10%	0.469	49.4%	51.2%	54.2%
20%	0.722	74.3%	75.8%	78%
30%	0.881	90.2%	91.4%	93.3%
40%	0.971	99.1%	100.3%	102%
50%	1	102%	103%	105%

(posledný riadok je úplne náhodný reťazec, kde každý bit je s rovnakou pravdepodobnosťou `0` alebo `1`, ktorý je nekomprimovateľný.)

Súvis s entropiou

Ak označíme n dĺžku celého bitvektoru, n_0 a n_1 počet núl, resp. počet jednotiek, tak všetky poradia zaberajú

$$\sum_i \left\lceil \lg \binom{r}{c_i} \right\rceil \leq \sum_i \lg \binom{r}{c_i} + n/r$$

a

$$\sum_i \lg \binom{r}{c_i} = \lg \prod_i \binom{r}{c_i} \leq \lg \binom{n}{n_1}.$$

Totiž $\prod_i \binom{r}{c_i}$ je počet spôsobov, ako vybrať c_1 pozícií jednotiek v prvom bloku a zároveň c_2 pozícií v druhom bloku a tak ďalej. Dokopy vyberieme n_1 pozícií z n, avšak každý blok má predpísaný počet jednotiek. Hodnota $\binom{n}{n_1}$ je proste počet výberov n_1 pozícií z n bez ďalších obmedzení, tzn. väčšie číslo.

Zo Stirlingovej aproximácie $\lg n! = n \lg n - n \lg e + O(\lg n)$ potom vyplýva

$$\begin{aligned}
\lg \binom{n}{n_1} &= \lg n! - \lg n_1! - \lg n_0! \\
&= n \lg n - \cancel{n \lg e} - n_1 \lg n_1 + \cancel{n_1 \lg e} - n_0 \lg n_0 + \cancel{n_0 \lg e} + O(\lg n) \\
&= n_1 \lg \frac{n}{n_1} + n_0 \lg \frac{n}{n_0} + O(\log n) \\
&= nH(S) + O(\log n)
\end{aligned}$$

Samozrejme, v RRR-kódovaní máme ešte navyše všetky c-čka, ktoré zaberajú $\lceil \lg(r+1) \rceil$ bitov pre každý blok a blokov je n/r. Spolu je to teda približne $n \times (\lg r)/r$, čo pre malé bloky nie je úplne zanedbateľné: pre $r = 31$ to je 16% z n, pre $r = 63$ takmer 10% z n.

V teórií si môžeme povedať, že zvolíme $r = \log n$ a tým pádom $n \times (\lg r)/r = o(n)$. V praxi na konštantách záleží zaujímajú nás hodnoty pre konkrétne n, nie chovanie keď $n \to \infty$.

Ako kódovať a dekódovať bloky

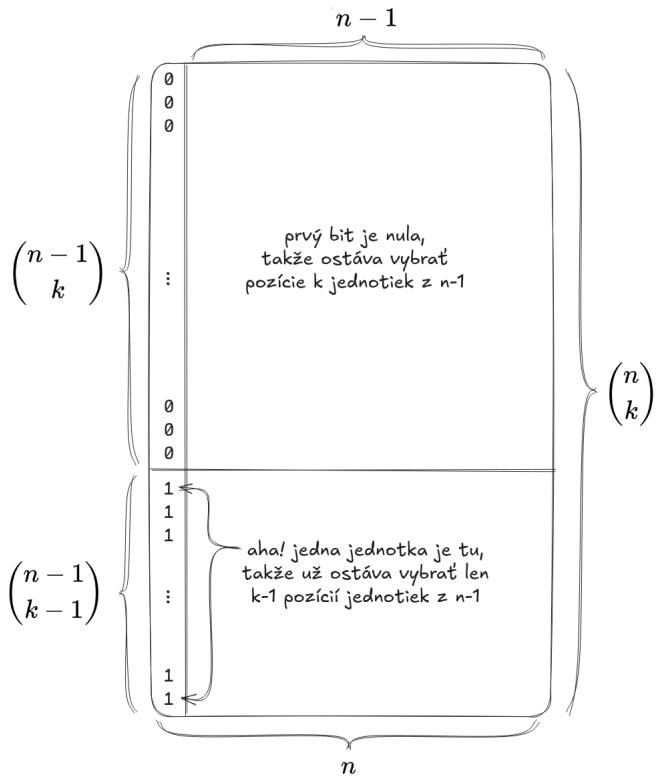
Poďme sa teraz pozrieť, ako implementovať jednotlivé operácie. V prvom rade budeme potrebovať vedieť jednotlivé bloky kódovať a dekódovať. Prezrite si ešte raz obrázok 1 vpravo. Otázka znie, ako napríklad vieme zistiť, že reťazec `0101100` je 15-ty (z 35) v poradí medzi reťazcami dĺžky 7 s 3-omi jednotkami? A naopak, ako vyzerá v poradí 23-tí reťazec dĺžky 7 s 3-omi jednotkami?

Skúste porozmýšľať sami a nájsť vhodný algoritmus na kódovanie a dekódovanie.

Ako pomôcka by vám malo stačiť nasledovné pozorovanie: Počet reťazcov dĺžky n s k jednotkami je presne $\binom{n}{k}$, pretože chceme vybrať k pozícií jednotiek z n. Sústreďme sa na prvý bit. Ten je buď `0` alebo `1`.

a) Ak je to `0`, stále ostáva vybrať k pozícií, ale zo zvyšných $n-1$ (to je $\binom{n-1}{k}$ možností).

b) Ak je to `1`, ostáva už vybrať iba $k-1$ pozícií zo zvyšných $n-1$ (to je $\binom{n-1}{k-1}$ možností).



To je známy vzorec pre kombinačné čísla

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1},$$

resp. známy fakt, že v Pascalovom trojuholníku je každé políčko rovné súčtu dvoch susedných políčok nad ním.

Keďže predpokladáme, že máme $\binom{n}{k}$ reťazcov zotriedených lexikograficky, tak prvých $\binom{n-1}{k}$ začína nulovým bitom a ďalších $\binom{n-1}{k-1}$ začína jednotkovým bitom.

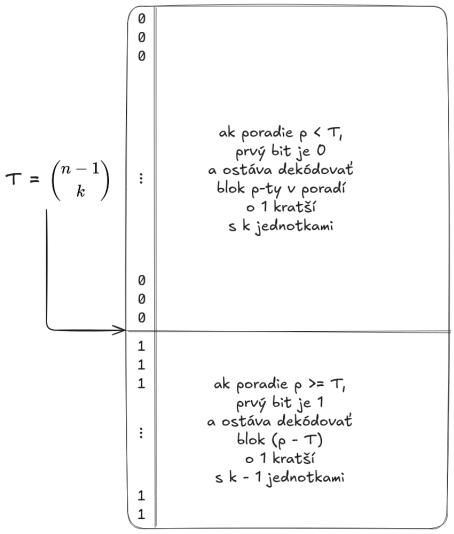
Riešenie je jednoduché, rekurzívne.

Dekódovanie. Príklad: Ktorý reťazec je 23-tí v poradí medzi reťazcami dĺžky 7 s 3-omi jednotkami? Spočítajme $\binom{6}{3} = 20$, tzn. prvých 20 reťazcov začína nulou (pamätajte, že poradie číslujeme od nuly). To znamená že 23-tí reťazec začína `1`, dokonca je tretí medzi tými, čo začínajú `1`, konkrétne tretí medzi reťazcami dĺžky 6 s 2-oma jednotkami. Dekódujeme zvyšok.

Spočítame $\binom{5}{2} = 10$, to znamená prvých 10 reťazcov začína na nulu, náš tretí reťazec je jeden z nich. Ďalší bit je `0`, dekódujeme zvyšok: tretí medzi reťazcami dĺžky 5 s 2-oma jednotkami.

Spočítame $\binom{4}{2} = 6$, ďalší bit je `0`, dekódujeme zvyšok: tretí medzi reťazcami dĺžky 4 s 2-oma jednotkami. Keďže $\binom{3}{2} = 3$ (a poradie počítame od nuly!), prvý bit je `1` a zvyšok je nultý v poradí – teda úplne na začiatku. Ostávajú 3 bity a jedna jednotka; na začiatku tohto zoznamu je `001`.

Výsledok: 23-tí v poradí je reťazec `1001001`. Pred ním je 20 reťazcov tvaru `0******` a 3 reťazce tvaru `1000***`.



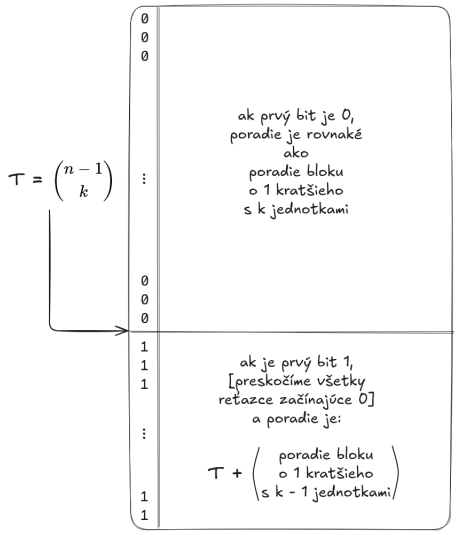
Kódovanie. Príklad: ako zakódujeme reťazec `0101100`? Nuž prvý bit je `0`, takže reťazec sa nachádza v prvej časti zoznamu (skôr ako $\binom{6}{3} = 20$). Prvý bit zahodíme a ostáva zakódovať 6-bitový reťazec `101100` s 3-oma jednotkami. Pred týmto reťazcom sú všetky tie, ktoré začínajú na nulu `0*****` – takých je

$\binom{5}{3} = 10$. To znamená, že `101100` sa nachádza za všetkými týmito desiatimi, až medzi reťazcami, čo začínajú na `1`. Koľkatý presne?

Zahodíme prvý bit a ostáva zistiť, koľkatý je reťazec `01100` – ten začína na nulu, takže poradie je rovnaké ako poradie `1100` dĺžky 4, s 2-oma jednotkami. Pred týmto sú všetky reťazce `0***` začínajúce na nulu, s 2-oma jednotkami a také sú 3. Tie preskočíme a ostane nám `100`, čo je v poradí druhý reťazec (číslujeme od 0!) medzi 3-bitovými s jednou jednotkou – konkrétne pred ním sú už iba `001` a `010`.

Výsledná pozícia `0101100` je teda $10 + 3 + 2 = 15$. Dostali sme ju vlastne tak, že sme spočítali, koľko reťazcov je *pred* ním a to sú:

- reťazce začínajúce `00*****`, ktorých je $\binom{5}{3} = 10$,
- reťazce začínajúce `0100***`, ktorých je $\binom{3}{2} = 3$, a
- reťazce začínajúce `01010**`, ktoré sú $\binom{2}{1} = 2$.

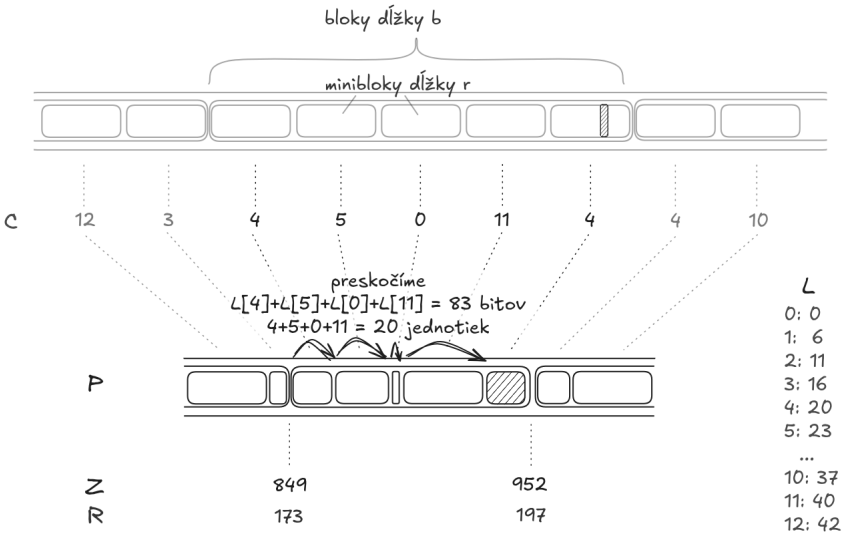


Výsledná štruktúra RRR

Budeme mať „minibloky“ dĺžky r, povedzme $r = 63$. Pri väčších miniblokoch treba počítať s tým, že kódovanie/dekódovanie bude trochu pomalšie, navyše, potrebujeme počítať s kombinačnými číslami a to najväčšie, $\binom{r}{r/2} \approx 2^r/\sqrt{\pi r/2}$ sa už nemusí zmestiť do 64-bitového registra, takže budeme potrebovať artimetiku s veľkými číslami.

Každý miniblok budeme kódovať ako dvojicu (c_i, p_i), kde c_i = #jednotiek v i-tom bloku a p_i = poradie medzi reťazcami dĺžky r s c jednotkami. Tieto hodnoty budeme mať uložené oddelene v dvoch poliach C a P.

V zhustenom poli C každá hodnota c_i zaberá $\lg(r+1)$ bitov, napríklad pre $r = 63$ je sú hodnoty 6-bitové. Takže do tohto poľa sa indexuje ľahko.



Obr. 2: Príklad štruktúry RRR. Úplne hore je zobrazený pôvodný vektor rozdelený na bloky a minibloky (len pre ilustráciu, po zakódovaní si ho, samozrejme, nemusíme pamätať). Celý RRR sa skladá z polí C – počty jednotiek v každom minibloku, P – kódy miniblokov, predstavujú poradie medzi reťazcami dĺžky r s c_i jednotkami, Z – pomocné pole pre indexy začiatkov blokov (keďže minibloky P majú rôzne dĺžky), R – predpočítané ranky, počet jednotiek po začiatok bloku, L – hodnoty $\lceil \lg \binom{r}{c} \rceil$.

Pole P obsahuje prvky s premenlivým počtom bitov. Budeme preto potrebovať pomocné pole, aby sme vedeli povedať, na ktorej pozícií začínajú bity zodpovedajúce k-temu bloku.

Celý bitvektor rozdelíme na väčšie bloky, povedzme dĺžky $b = 16 \times 63 = 1008$. Pre každý začiatok bloku si spočítame rank $R[i]$ – koľko jednotiek bolo doteraz a pozíciu $Z[i]$, kde začína zakódovaný i-ty blok.

Okrem toho sa nám pri výpočtoch bude hodiť predpočítaná tabuľka kombinačných čísiel (Pascalov trohuholník) a predpočítané dĺžky kódov pre daný počet jednotiek c_i – t.j. hodnoty $L[c] = \lceil \lg \binom{r}{c} \rceil$.

Ak teraz budeme chcieť napríklad zistiť hodnotu i-teho bitu, nech

$$\ell = \underbrace{\lfloor i/b \rfloor}_{\text{číslo bloku}}, \qquad k = \underbrace{\lfloor (i - \ell b)/r \rfloor}_{\text{číslo minibloku v rámci bloku}}, \qquad j = \underbrace{i \bmod r}_{\text{index v rámci minibloku}}.$$

Potom i-ty bit je j-ty bit v rámci k-teho minibloku vnútri ℓ-tého bloku. V tabuľke $Z[\ell]$ nájdeme, kde začína ℓ-tý blok. Potom preskočíme k-miniblokov – ich dĺžky vieme tak, že čítame pole C, kde je počet jednotiek a z poľa L sa dozvieme, aká dĺžka kódu tomu zodpovedá. Tak sa dostaneme ku k-temu minibloku, ten dekódujeme a prečítame j-ty bit.

Ak chceme zistiť $\text{rank}_1(i)$, postupujeme podobne – v tabuľke $R[\ell]$ nájdeme

rank po ℓ-tý blok. Ako preskakujeme k-miniblokov, zároveň si nasčítavame hodnoty C, teda počet jednotiek v týchto miniblokoch. Nakoniec k-ty blok dekódujeme a spočítame rank po j-ty bit.