APLIKACIJE RAČUNALNIŠKIH ALGORITMOV

Borut Žalik

UNIVERZA V MARIBORU
FAKULTETA ZA ELEKTROTEHNIKO,
RAČUNALNIŠTVO IN INFORMATIKO

Aplikacije računalniških algoritmov

Borut Žalik

MARIBOR, 2023

Naslov *Title* — **Aplikacije računalniških algoritmov**
Applications of Computer Algorithms

Avtor *Author* — Borut Žalik
(Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko)

Recenzija *Review* — Marjan Mernik
(Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko)

Saša Divjak
(Univerza v Ljubljani Fakulteta za računalništvo in informatiko)

Štefan Kohek
(Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko)

Jezikovni pregled *Language edeting* — TAIA INT d.o.o.

Tehnična urednika *Technical editors* — Borut Žalik
(Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko)

Jan Perša
(Univerza v Mariboru, Univerzitetna založba)

Grafične priloge *Graphics material* — Borut Žalik, 2023

Oblikovanje ovitka *Cover designer* — Tine Matjašič

Grafika na ovitku *Cover graphic* — Borut Žalik, 2023

Založnik *Published by* — **Univerza v Mariboru**
Univerzitetna založba
Slomškov trg 15, 2000 Maribor, Slovenija
https://press.um.si, zalozba@um.si

Izdajatelj *Issued by* — **Univerza v Mariboru**
Fakulteta za elektrotehniko, računalništvo in informatiko
Koroška cesta 46, 2000 Maribor, Slovenija
https://feri.um.si, feri@um.si

Izdaja *Edition* — Prva izdaja

Izdano *Published at* — Maribor, junij 2023

Vrsta publikacije *Publication type* — E-knjiga

Dostopno na *Available at* — https://press.um.si/index.php/ump/catalog/book/787

```
CIP - Kataložni zapis o publikaciji
Univerzitetna knjižnica Maribor

004.021(0.034.2)

ŽALIK, Borut
    Aplikacije računalniških algoritmov
[Elektronski vir] / avtor Borut Žalik. -
1. izd. - E-publikacija. - Maribor :
Univerza v Mariboru, Univerzitetna
založba, 2023

Način dostopa (URL):
https://press.um.si/index.php/ump/catalog/
book/787
ISBN 978-961-286-752-2 (Web, PDF)
doi: 10.18690/um.feri.6.2023
COBISS.SI-ID 156306435
```

© Univerza v Mariboru, Univerzitetna založba
/ *University of Maribor, University Press*

Besedilo / *Text* © Žalik, 2023

To delo je objavljeno pod licenco Creative Commons Priznanje avtorstva 4.0 Mednarodna.
/ *This work is licensed under the Creative Commons Attribution 4.0 International License.*

Uporabnikom je dovoljeno tako nekomercialno kot tudi komercialno reproduciranje, distribuiranje, dajanje v najem, javna priobčitev in predelava avtorskega dela, pod pogojem, da navedejo avtorja izvirnega dela.

Vsa gradiva tretjih oseb v tej knjigi so objavljena pod licenco Creative Commons, razen če to ni navedeno drugače. Če želite ponovno uporabiti gradivo tretjih oseb, ki ni zajeto v licenci Creative Commons, boste morali pridobiti dovoljenje neposredno od imetnika avtorskih pravic.

https://creativecommons.org/licenses/by-nc-nd/4.0/

ISBN 978-961-286-752-2 (pdf)
978-961-286-753-9 (mehka vezava)

DOI https://doi.org/10.18690/um.feri.6.2023

Cena *Price* — Brezplačni izvod

Odgovorna oseba založnika *For publisher* — prof. dr. Zdravko Kačič,
rektor Univerze v Mariboru

Citiranje *Attribution* — Žalik, B. (2023). *Aplikacije računalniških algoritmov*. Univerza v Mariboru, Univerzitetna založba. doi: 10.18690/um.feri.6.2023

Kazalo

1 Urejanje podatkov v linearnem času **3**
1.1 Števno urejanje 4
1.2 Urejanje Roman 8
1.3 Korensko urejanje 9
1.4 Urejanje z vedri 9

2 Iskanje vzorca v zaporedju **13**
2.1 Naivni pristop 14
2.2 Rabin-Karpov algoritem 14
2.3 Knut-Morris-Prattov algoritem 17
2.4 Horspoolov algoritem 20
2.5 Sundayev algoritem 22

3 Najkrajša razdalja urejanja zaporedij **25**
3.1 Wagner-Fischerjev algoritem 28

4 Preprosti algoritmi šifriranja **31**
4.1 Pomikalni šifrirnik 31
4.2 Vigenérjev šifrirnik 33
4.2.1 Šifrirnik Playfair 34
4.2.2 Šifrirnik ADFGX 36

5 Stiskanje podatkov **41**
5.1 Intuitivne metode stiskanja 43
5.1.1 Stiskanje zaporedja enakih znakov 45
5.2 Entropija informacije 46
5.3 Statistično stiskanje podatkov 50
5.3.1 Shannon-Fanojev algoritem 51
5.3.2 Huffmanov algoritem 53
5.3.3 Huffmanov algoritem s prilagajanjem 58

5.3.4 Poenostavljen algoritem stiskanja s prilagajanjem . . . 64
5.4 Aritmetično kodiranje 66
5.4.1 Ideja aritmetičnega kodiranja 66
5.4.2 Celoštevilske implementacije aritmetičnega kodiranja . 70
5.4.2.1 Implementacija aritmetičnega kodiranja s pomikanjem 71
5.4.2.2 Aritmetično kodiranje s skaliranjem 75
5.5 Stiskanje podatkov s slovarjem 83
5.5.1 Algoritem LZ77 85
5.5.2 Algoritem LZSS 86
5.5.3 Algoritem LZ78 89
5.5.4 Algoritem LZW 91
5.6 Kodiranje zaporedij celih števil 94
5.6.1 Golombovo kodiranje 94
5.6.2 Golomb-Riceovo kodiranje 97
5.6.3 Prilagodljivo binarno zaporedno kodiranje 98
5.6.4 Prilagodljivo binarno zaporedno kodiranje z vrnitvijo . 101
5.6.5 Interpolativno kodiranje 102
5.6.5.1 FELICS 106

6 Metode transformacije zaporedij 113
6.1 Transformacija premik naprej 113
6.2 Transformacija inverzne frekvence 116
6.3 Transformacija kodiranje razdalj 119
6.4 Transformacija desno manjše 120
6.5 Transformacija z drevesom valčkov 122
6.6 Burrows-Wheelerjeva transformacija 126

7 Priponska polja in priponska drevesa 131
7.1 Priponsko polje 131
7.1.1 Algoritem DC3 133
7.2 Številsko in priponsko drevo 143
7.2.1 Konstrukcija priponskega drevesa z naivno metodo . . 146
7.2.2 Ukkonenov algoritem 148
7.2.3 Realizacija Ukkonenovega algoritma 153
7.2.4 Uporaba priponskih dreves 160

8 Pretvorba večdimenzionalnih podatkov v zaporedja 165
8.1 Krivulje polnjenja prostora 165
8.2 Konstrukcija Hilbertove krivulje 167

8.2.1 Predstavitev Hilbertove krivulje z drevesom 169
8.3 Transformacijo v in iz prostora Hilbertove krivulje z diagramom stanj 171
8.4 Verižne kode 174
8.4.1 Freemanova verižna koda v osem smeri 174
8.4.2 Freemanova verižna koda v štiri smeri 175
8.4.3 Izpeljanke Freemanove verižne kode 175
8.4.4 Ogliščna verižna koda 177
8.4.5 Triortogonalna verižna koda 178
8.4.6 Središčno-lomna verižna koda 179
8.4.7 Nepredznačena verižna koda Manhattan 180

Seznam najpogostejših simbolov

$H(I)$	Shannonova entropija informacije
$\mathcal{I}$	interval
I, J	vhodni niz/zaporedje/polje
$\lvert I\rvert, \lvert O\rvert$	dolžina niza/zaporedja/polja
O	izhodni niz/zaporedje/polje
$\lvert\Sigma\rvert$	moč abecede
Σ	abeceda
σ_i	element abecede
$\mathcal{T}$	drevo

PREDGOVOR

Vaja dela mojstra.

Človeško zgodovino tako ali drugače zaznamuje **znanje**. Znanje je bilo včasih dostopno le izbrancem, hranjeno v templjih, samostanih, redkih knjižnicah in prvih izobraževalnih središčih. Zato je pri razvoju novega znanja lahko sodelovalo zelo malo ljudi. Posledično je bil razvoj počasen, pogosto temelječ na popolnoma zgrešenih dogmatskih teorijah.

Raziskave na področjih fizike, materialov, elektronike, telekomunikacij, elektrotehnike in matematike so šele pred nekaj desetletji omogočile revolucionarni prelom; znanje, shranjeno na magnetnih in elektronskih nosilcih, je bilo možno učinkovito prenesti in prikazati uporabnikom na razumljiv način. S tem je akumulirano znanje človeštva postalo dostopno večini ljudi v vsakem trenutku na vsaki točki našega planeta. Najpomembneje pa je, da dostop do znanja ni diskriminatoren in da je zelo poceni. V globalno zakladnico človeškega znanja lahko danes prispeva veliko več Zemljanov, zato količina znanja narašča hitreje kot kadarkoli do sedaj. Tega razkošja ste deležni tudi sami, cenjene študentke in študenti, čim bolje ga izkoristite.

Organizacija in shranjevanje znanja, njegovo učinkovito kodiranje, iskanje, prikazovanje, zlivanje ter samodejno tvorjenje novega znanja je privedlo do rojstva nove znanstvene discipline – **računalništva** (angl. Computer Science). V svojem bistvu je računalništvo umetnost določanja lastnosti **univerzalnega stroja**, ki ga danes imenujemo **računalnik**. Univerzalni stroj za spremembo svojega delovanja ne zahteva spremembe strukture stroja, ne zahteva novega načrtovanja in novega proizvodnega procesa, dovolj je, da spremenimo zaporedje ukazov, ki jih stroj zna izvajati. Posledično je izdelava univerzalnega stroja poceni, njegova uporabnost pa ni omejena. Ključna paradigma univerzalnega stroja je spreminjaje zaporedja njegovih ukazov, čemur pravimo **programiranje**, zaporedju ukazov, ki opravijo zadano nalogo, pa **program**.

Programiranje se je kmalu odmaknilo od fizične ravni elementarnih ukazov. Danes programiramo na višji ravni abstrakcije, a zavedanje o načinu delovanja računalnika nam omogoča realizacijo učinkovitejših programov. Človeštvo se s programiranjem srečuje šele nekaj zadnjih desetletij, zato se je programiranja treba **naučiti**. V Sloveniji žal temu področju v splošnem izobraževanju ne namenjamo nobene pozornosti, s čimer krnimo splošno izobrazbo in konkurenčno sposobnost otrok.

Vsako učenje, vsak trening, zahteva napor in vajo. Tudi pri programiranju ni drugače. Učenje in trening pa sta veliko uspešnejša, če v njiju vidimo smisel in če začutimo svoj napredek. Prav treningu programiranja in postopnemu nadgrajevanju težavnosti algoritmov je namenjen pričujoč učbenik. Nastal je na podlagi predavanj pri predmetih *Aplikacije računalniških algoritmov* v prvem letniku univerzitetnega študijskega programa Računalništvo in informacijske tehnologije in *Algoritmi v računalniški praksi* v drugem letniku visokošolskega študijskega programa Računalništvo in informacijske tehnologije na Fakulteti za elektrotehniko, računalništvo in informatiko Univerze v Mariboru (UM FERI). Z izbranimi vsebinami smo po eni strani želeli študentom približati zavedanje o uporabnosti programiranja, po drugi strani pa izbrati teme, ki so primerne za prve ambicioznejše programerske podvige.

O algoritmih obstaja ogromno knjig, a ta učbenik se ne zgleduje po nobeni od njih. S teoretično analizo, dokazovanjem pravilnosti algoritmov in strategijami delovanja se ne ukvarjamo. Ideje algoritmov raje razložimo neformalno z ilustrativnimi primeri. Posledično lahko predstavimo več algoritmov, ki naj bodo navdih študentom, da bi jih programirali, morebiti vključili v svoje aplikacije in s tem dvignili svojo programersko učinkovitost. Temu je namenjen tudi izbor vsebin, ki se jih v učbeniku dotaknemo, in sicer:

- urejanje v linearnem času,
- iskanje vzorcev v nizih,
- iskanje najkrajše razdalje zaporedij,
- preprosti šifrirniki,
- stiskanje podatkov,
- transformacije nizov,
- priponska polja in priponska drevesa ter
- algoritmi v rastrskem prostoru (krivulje polnjenja prostora in verižne kode).

Da bi celoviteje zajeli določeno problematiko, najdemo v učbeniku tudi nekoliko zahtevnejše algoritme. Za njihovo popolno razumevanje bo študent moral poseči po dodatni literaturi, kar pa je tudi namen študija.

Vsako poglavje zaključuje nabor vprašanj in predlogov za realizacijo, s čimer omogočimo študentom, da se pripravijo na izpit, ob tem izpopolnijo svoje programerske veščine in se pri programiranju tudi zabavajo.

Poglavje 1

Urejanje podatkov v linearnem času

Urejanje zaporedja elementov je med najbolj klasičnimi nalogami v programiranju, za katero obstaja množica postopkov, kot so: urejanje z mehurčki (angl. bubble sort), urejanje z izbiranjem (angl. selection sort), urejanje z vstavljanjem (angl. insertion sort), urejanje s kopico (angl. heap sort), hitro urejanje (angl. quick sort), urejanje z zlivanjem (angl. merge sort). Nekateri od navedenih postopkov zaporedje uredijo v času $O(n^2)$, drugi celo v času $O(n \log n)$, pri čemer je n dolžina zaporedja.

Pri urejanju je pogosto pomembna tudi količina pomnilnika, ki ga algoritem potrebuje za svoje delovanje. Če je zahteva po dodatnem pomnilniku enaka, ne glede na velikost zaporedja, ki ga urejamo, pravimo, da algoritem **ureja podatke na mestu** (angl. in-place sort).

Pomembna lastnost algoritmov urejanja je tudi **stabilnost**. Zaporedje naj bo $I = \langle \sigma_i \rangle$, $0 \leq i < n$. Elementa $(\sigma_u, \sigma_v) \in I$ naj bosta enaka ($\sigma_u = \sigma_v$) in naj velja $u < v$. Urejanje je stabilno, če po urejanju vedno velja $u < v$. Poglejmo si primer 1.1a, kjer so elementi σ_i imena. Vidimo, da so na položajih $i = 1$, $i = 3$ in $i = 5$ imena, ki se začnejo s črko A. Uredimo zaporedje I po prvi črki abecede. Algoritmi stabilnega urejanja bodo vrnili rezultat, ki ga kaže primer 1.1b. Če bi bilo urejanje nestabilno, bi lahko bila imena Aljaž, Andraž in Anže na katerem koli položaju $i = \{0, 1, 2\}$.

Urejanje podatkov opravimo z zamenjavami vrednosti v zaporedju, kar realiziramo z začasnim odlagališčem, tako da za zamenjavo dveh elementov potrebujemo tri zamenjave. Najprej je videti, da se zamenjavi podatkov ne moremo izogniti, a v nekaterih primerih lahko zaporedje uredimo tudi drugače. Prav takšne algoritme si bomo ogledali v nadaljevanju.

i	I	O
0	Drago	Aljaž
1	Aljaž	Andraž
2	Vesna	Anže
3	Andraž	Drago
4	Jelka	Iztok
5	Anže	Jelka
6	Iztok	Vesna
	(a)	(b)

Primer 1.1: (a) Vhodno zaporedje I, (b) stabilno urejeno zaporedje O

1.1 Števno urejanje

Kot smo omenili, je bistvo vseh metod urejanja primerjava dveh vrednosti in, glede na izid primerjave, tudi njuna zamenjava. A ideja števnega urejanja (angl. counting sort) je popolnoma drugačna [1]. Namesto primerjav raje **preštejemo** enake vrednosti in jih nato pravilno umestimo v urejeno zaporedje.

Naj bo $I = \langle \sigma_\mathrm{i} \rangle$, $0 \leq i < n$, zaporedje, ki ga želimo urediti, in $n = |I|$ njegova dolžina. I sestoji iz simbolov, ki pripadajo abecedi $\Sigma = \{\sigma_i\}$. Števno urejanje bo rezultat zapisalo v zaporedje $O = \langle \sigma_\mathrm{j} \rangle$, $\sigma_j \leq \sigma_{j+1}$, $0 \leq j < n-1$, $\sigma_j \in \Sigma$, $|I| = |O|$. Algoritem potrebuje za delovanje še pomožno zaporedje $C = \langle c_i \rangle$ z dolžino $|C| = |\Sigma|$. Postopek urejanja razložimo s primerom 1.2. Najprej pripravimo zaporedji O in C.

$$
\begin{aligned}
i:&\quad \texttt{0 1 2 3 4 5 6 7} \\
I =& \langle \texttt{b d c a b c a c} \rangle \\
O =& \langle \texttt{- - - - - - - -} \rangle \\
\\
\Sigma =& \{ \texttt{a b c d} \} \\
C =& \langle \texttt{0 0 0 0} \rangle
\end{aligned}
$$

Primer 1.2: Inicializacija števnega urejanja

Algoritem nato potuje skozi zaporedje I. Za vsak $\sigma_i \in I$ določi njegov zaporedni položaj v abecedi Σ (položaj označimo z i_Σ), nato pa inkrementira števec v zaporedju C, $C_{i_\Sigma} = C_{i_\Sigma} + 1$. Stanje v zaporedjih po prvem koraku algoritma kaže primer 1.3. Polje C (angl. counter) očitno hrani število ponovitev/frekvenco vsakega simbola $\sigma_i \in I$.

V drugem koraku po vrsti obiščemo vse elemente zaporedja C ter k

$$
\begin{aligned}
i:&\quad \texttt{0 1 2 3 4 5 6 7}\\
I=&\langle\texttt{b d c a b c a c}\rangle\\
O=&\langle\texttt{- - - - - - - -}\rangle\\
\\
\Sigma=&\{\texttt{a b c d}\}\\
C=&\langle\texttt{2 2 3 1}\rangle
\end{aligned}
$$

Primer 1.3: Prvi korak števnega urejanja

trenutnemu elementu prištejemo vrednost predhodnega, $C_i = C_i + C_{i-1}$, $0 < i < |\Sigma|$, s čimer zgradimo zaporedje **kumulativnih vsot ponovitev** vrednosti (glej primer 1.4).

$$
\begin{aligned}
i:&\quad \texttt{0 1 2 3 4 5 6 7}\\
I=&\langle\texttt{b d c a b c a c}\rangle\\
O=&\langle\texttt{- - - - - - - -}\rangle\\
\\
\Sigma=&\{\texttt{a b c d}\}\\
C=&\langle\texttt{2 4 7 8}\rangle
\end{aligned}
$$

Primer 1.4: Drugi korak števnega urejanja

V zadnjem (tretjem) koraku tvorimo urejeno zaporedje O. Zaporedje I obiskujemo od zadnjega elementa proti prvemu. Za trenutni element $\sigma_i \in I$ najdemo njegov indeks i_Σ v abecedi ter ga uporabimo kot kazalec v zaporedje C. Vrednost $C_{i_\Sigma} - 1$ kaže položaj v zaporedju O, kamor moramo vpisati σ_i. Po vpisu dekrementiramo vrednost v C. Razložimo s primerom 1.4. Zadnji element $I_7 = \texttt{c}$, ki je tretji element abecede Σ, torej $i_\Sigma = 2$. Ker je $C_{i_\Sigma=2} = 7$, vpišemo v $O_{7-1} = \texttt{c}$. Vrednost C_2 zatem zmanjšamo za ena. Situacijo kaže primer 1.5.

$$
\begin{aligned}
i:&\quad \texttt{0 1 2 3 4 5 6 7}\\
I=&\langle\texttt{b d c a b c a c}\rangle\\
O=&\langle\texttt{- - - - - - c -}\rangle\\
\\
\Sigma=&\{\texttt{a b c d}\}\\
C=&\langle\texttt{2 4 6 8}\rangle
\end{aligned}
$$

Primer 1.5: Stanje po postavitvi elementa iz I_7 na pravo mesto v O

Nadaljujemo z naslednjim elementom, to je $I_6 = \texttt{a}$, ki je prvi element abecede ($i_\Sigma = 0$). Ker je $C_0 = 2$, je $O_{2-1=1} = \texttt{a}$. Zatem C_0 dekrementiramo

(glej primer 1.6). Na ta način algoritem nadaljuje do prvega elementa v I, ko dobimo stanje, ki ga kaže primer 1.7.

$$\begin{array}{rl}
i: & \texttt{0 1 2 3 4 5 6 7} \\
I = & \langle \texttt{b d c a b c a c} \rangle \\
O = & \langle \texttt{- a - - - - c -} \rangle \\
\\
\Sigma = & \{\texttt{a b c d}\} \\
C = & \langle \texttt{1 4 6 8} \rangle
\end{array}$$

Primer 1.6: Postavitev naslednjega elementa iz I na pravo mesto v O

$$\begin{array}{rl}
i: & \texttt{0 1 2 3 4 5 6 7} \\
I = & \langle \texttt{b d c a b c a c} \rangle \\
O = & \langle \texttt{a a b b c c c d} \rangle \\
\\
\Sigma = & \{\texttt{a b c d}\} \\
C = & \langle \texttt{0 2 4 7} \rangle
\end{array}$$

Primer 1.7: Stanje po zaključenem števnem urejanju

Če urejamo števila (roko na srce, tudi znakom iz Σ v primeru 1.2 bi lahko enolično priredili števila), je implementacija števnega urejanja še preprostejša. Tokrat naj bo $\Sigma \in \mathbb{Z}$. Najprej določimo interval $[l, r]$, $l < r$, na katerem so $\sigma_i \in I$. Elemente zaporedja I nato premaknemo na interval $[0, r - l]$ ter postavimo $|C| = l - r + 1$.

Imejmo primer 1.8. Določimo interval $[l = -2, r = 9]$, ki ga hkrati z elementi iz I premaknemo na interval $[0, 11]$ ter postavimo $|C| = 12$.

$$I = \langle \texttt{-2 7 9 -2 -1 0 9 7 9 -2} \rangle$$

Primer 1.8: Zaporedje števil, ki ga bomo uredili s števnim urejanjem

$$\begin{array}{rl}
i: & \texttt{0 1 2 3 4 5 6 7 8 9 10 11} \\
I = & \langle \texttt{0 9 11 0 1 2 11 9 11 0} \rangle \\
O = & \langle \texttt{- - - - - - - - - -} \rangle \\
C = & \langle \texttt{3 1 1 0 0 0 0 0 0 2 0 3} \rangle
\end{array}$$

Primer 1.9: Stanje po premiku vrednosti $\sigma_i \in I$ na interval $[0, 11]$ in polnjenju zaporedja C

$$
\begin{array}{rl}
i: & \texttt{0 1 2 3 4 5 6 7 8 9 10 11} \\
I = & \langle \texttt{0 9 11 0 1 2 11 9 11 0} \rangle \\
O = & \langle \texttt{- - - - - - - - - -} \rangle \\
C = & \langle \texttt{3 4 5 5 5 5 5 5 5 7 7 10} \rangle
\end{array}
$$

Primer 1.10: Določitev kumulativnih vsot ponovitev v C

Po prvem koraku algoritma imamo situacijo, ki jo kaže primer 1.9. V drugem koraku algoritem v zaporedje C izračuna kumulativne vrednosti (glej primer 1.10). Stanje po zaključku algoritma števnega urejanja kaže primer 1.11. Ob koncu premaknemo elemente v O na začetni številski interval; v našem primeru od vsakega elementa odštejemo vrednost 2.

$$
\begin{array}{rl}
i: & \texttt{0 1 2 3 4 5 6 7 8 9 10 11} \\
I = & \langle \texttt{0 9 11 0 1 2 11 9 11 0} \rangle \\
O = & \langle \texttt{0 0 0 1 2 9 9 11 11 11} \rangle \\
C = & \langle \texttt{0 3 4 5 5 5 5 5 5 5 7 7} \rangle
\end{array}
$$

Primer 1.11: Stanje po zadnjem koraku števnega urejanja pred vrnitvijo vrednosti na začetni interval

Števno urejanje je stabilno urejanje. Hitro ugotovimo, da implementacija deluje v linearnem času. Poglejmo:

- v času $O(n)$ najdemo interval $[l, r]$;
- v času $O(n)$ premaknemo vrednosti v I na interval $[0, l - r]$;
- v času $O(n)$ preštejemo pogostost pojavljanja elementov $\sigma_i \in I$;
- v času $O(\Delta)$, $\Delta = r - l$, tvorimo kumulativne vsote ponovitev v C;
- v času $O(n)$ vpišemo elemente v O in
- v času $O(n)$ premaknemo elemente v O na interval $[l, r]$.

Časovna zahtevnost je $5\,O(n) + O(\Delta)$, kjer ločimo med tremi možnostmi:

- če je $n \approx \Delta$, potem je časovna zahtevnost $6\,O(n) = O(n)$;
- če je $n >> \Delta$, je časovna zahtevnost $5\,O(n) = O(n)$;
- če je $n << \Delta$, je časovna zahtevnost $O(\Delta)$.

Ključna omejitev števnega urejanja je velikost abecede. Zaporedij, ki jih ne bi znali preslikati v indekse polja C, ali zaporedij, katerih abeceda bi presegla pomnilniške sposobnosti računalniškega stroja, nam ne uspe urediti (na primer števila s plavajočo vejico). V takšnem primeru moramo uporabiti druge algoritme urejanja. Izboljšavo števnega urejanja najdemo v [2], paralelno izvedbo na grafičnih procesnih enotah (angl. graphics processing unit, GPU) s CUDA pa v [3].

1.2 Urejanje Roman

Zanimiva izpeljanka števnega urejanja je urejanje Roman, ki ga je predlagal Roman Čuk [4] med študijem na visokošolskem študiju računalništva in informatike na UM FERI. Roman je močno poenostavil števno urejanje. Zmanjšal je število korakov; zaporedja O, kamor vpisujemo urejeno zaporedje, ne potrebujemo več; urejeno zaporedje zapišemo kar v I.

Prvi korak algoritma je enak kot pri števnem urejanju; preštejemo elemente in njihovo število shranimo v C. S tem imamo tudi dovolj informacij, da lahko generiramo urejeno zaporedje. Po vrsti se od leve proti desni premikamo skozi C in v I vpišemo toliko vrednosti, kot jih hrani C_i. Postopek razjasnimo s primerom 1.12, kjer vidimo situacijo po prvem koraku. Ker je $C_0 = 0$, v I ne vpišemo ničesar. $C_1 = 2$ nam pove, da v I vpišemo dve enici na položaja I_0 in I_1. $C_2 = 0$, zato v I ne vpišemo ničesar. Ker je $C_3 = 2$, na položaja I_2 in I_3 vpišemo trojki. Trenutno stanje kaže primer 1.13. Ko obiščemo vse elemente polja C, bo polje I hranilo urejeno zaporedje.

$$
\begin{aligned}
i:&\quad \mathtt{0\ 1\ 2\ 3\ 4\ 5\ 6\ 7}\\
I=&\langle \mathtt{3\ 6\ 4\ 1\ 3\ 4\ 1\ 4}\rangle\\
C=&\langle \mathtt{0\ 2\ 0\ 2\ 3\ 0\ 1\ 0}\rangle
\end{aligned}
$$

Primer 1.12: Urejanje Roman po prvem koraku

$$
\begin{aligned}
i:&\quad \mathtt{0\ 1\ 2\ 3\ 4\ 5\ 6\ 7}\\
I=&\langle \mathtt{1\ 1\ 3\ 3\ 3\ 4\ 1\ 4}\rangle\\
C=&\langle \mathtt{0\ 2\ 0\ 2\ 3\ 0\ 1\ 0}\rangle
\end{aligned}
$$

Primer 1.13: Stanje, ko smo vstavili vrednosti iz C_0, C_1, C_2 in C_3

1.3 Korensko urejanje

Korensko urejanje (angl. radix sort) je zelo stara metoda urejanja. Izumil jo je Herman Hollerith in zanjo leta 1889 pridobil tudi patent [5]. V tistih časih računalnika seveda še niso poznali, zato je Hollerith izdelal kar celo napravo, s katero je sešteval informacije, shranjene na luknjastih karticah. Njegovo idejo hitro pretvorimo v delujoč algoritem, ki si ga oglejmo s primerom 1.14. Zaradi lažje ponazoritve delovanja algoritma bomo vhodno zaporedje I zapisali navpično.

I	1. prehod	2. prehod	3. prehod
219	244	219	219
346	325	619	244
546	346	325	325
728	546	728	346
325	728	244	546
619	219	346	619
244	619	546	728

Primer 1.14: Korensko urejanje

Pri prvem prehodu uredimo števila glede na zadnjo števko (pravimo ji ključ). V drugem koraku uredimo ključe glede na desetice, pri čemer pa ne smemo premešati enakih ključev iz prvega koraka, torej enic. Zato potrebujemo **stabilen** algoritem urejanja (na primer števno urejanje). Do zadnjega koraka je zaporedje neurejeno, nato pa se skoraj čudežno uredi, ko, v našem primeru, uredimo še stotice. Očitno potrebuje korensko urejanje d-prehodov, pri čemer je d število števk največjega števila, ki ga urejamo. Če uredimo ključe v linearnem času, korensko urejanje deluje v linearnem času. Korensko urejanje lahko hitro priredimo tudi za urejanje števil s plavajočo vejico [6] in paralelno urejanje tako na večjedrnih procesorjih [7] kot na GPU [8].

1.4 Urejanje z vedri

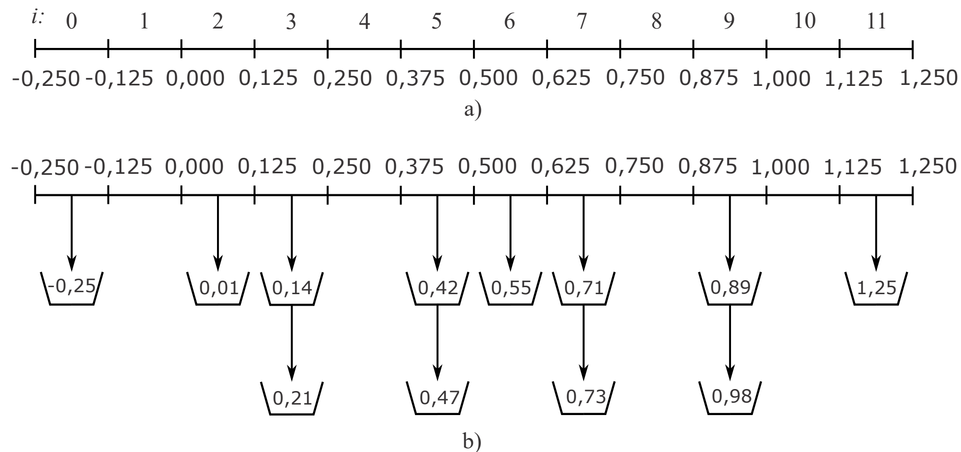
Pri **urejanju z vedri** (angl. bucket sort, bin sort) porazdelimo elemente $\sigma_i \in I$ na **sezname veder** (angl. buckets) [1], vsak seznam nato uredimo posebej. Zatem po vrsti obiščemo elemente v seznamu veder in dobimo urejeno zaporedje. Poglejmo si primer 1.15.

$I = \langle$`0,14 0,73 0,98 −0,25 0,42 0,01 0,21 0,47 1,25 0,89 0,55 0,71`$\rangle$

Primer 1.15: Vhodno zaporedje za urejanje z vedri

Najprej s sprehodom skozi I določimo interval $[l, r]$, znotraj katerega so vsi σ_i. V našem primeru je interval $[-0{,}25,\ 1{,}25]$. Interval razdelimo na n delov. Ker je $n = |I| = 12$, dobimo situacijo na sliki 1.1a. Zatem $\forall \sigma_i \in I$ z enačbo 1.1 izračunamo indeks i podintervala, ki določa seznam veder, kamor bomo vstavili σ_i. Na primer, $\sigma_0 = 0{,}14$, enačba 1.1 vrne $i = 2$, kamor vstavimo σ_0. Sezname veder, ki vsebujejo več kot eno vedro, nato uredimo s kakšnim drugim algoritmom ali z rekurzivnim klicem urejanja z vedri. Dobimo situacijo, ki jo vidimo na sliki 1.1b. Urejeno zaporedje sestavimo z zaporednim sprehodom skozi vse sezname veder.

$$i = \left\lfloor n \frac{\sigma_i - l}{r - l} \right\rfloor \tag{1.1}$$



Slika 1.1: Primer urejanja z vedri

Razmislimo, ali si urejanje veder sploh zasluži biti član poglavja algoritmov urejanja, ki delujejo v linearnem času. Odgovor je pritrdilen, če izpolnimo naslednjima pogojema:

- število podintervalov mora biti približno enako dolžini zaporedja, $n \approx |I|$, in
- porazdelitev vrednosti elementov v I mora biti vsaj blizu enakomerne/uniformne porazdelitve.

Prvega pogoja danes ni težko zagotoviti, saj so računalniški stroji opremljeni z Gi zlogi pomnilnika, kar je več kot dovolj za običajne aplikacije. Drugi pogoj pa je zahtevnejši. Kakšna je porazdelitev vrednosti $\sigma_i \in I$, praviloma ni znano. Šele ko opravimo polnjenje veder, lahko ugotovimo, da so nekateri seznami veder zelo polni, številni pa so praznih. Skrajni možnost kaže primer 1.16. Tokrat bi postavili $n = 10$. Vsi elementi, razen prvega, bi bili uvrščeni na zadnji seznam veder, kjer bi morali urediti $|I|-1$ elementov. To nalogo lahko rešimo na več načinov, najugodnejša pa je z rekurzivnim klicanjem urejanja z vedri. Ob ustrezni implementaciji je tako urejanje z vedri še vedno zelo učinkovito.

$$I = \langle \texttt{0,01 0,92 0,98 0,94 0,99 0,92 0,95 0,97 0,96 0,92} \rangle$$

Primer 1.16: Neugodna porazdelitev podatkov za urejanje z vedri

Pri neenakomerni porazdelitvi vhodnih podatkov se lotimo algoritma na naslednji način:

- Pri prvem prehodu razvrstimo podatke v vedra, kot smo opisali.
- Pregledamo zaporedje veder in vsako zaporedje, ki vsebuje več kot k elementov, rešimo rekurzivno.
- Ko je $k \leq 2$, zaporedje uredimo trivialno in prekinemo rekurzijo.

Analize urejanja z vedri najdemo v [9, 10, 11], vzporedno izvedbo algoritma pa v [12].

Naloge

1. Razmislite, ali je vsem dobro znan algoritem hitrega urejanja stabilen.
2. Razmislite, v katerih primerih je pomembno, da algoritem ureja podatke stabilno.
3. Naj bo $I = \langle \mathsf{3, 5, -4, 9, 10, 10, 5, 4, -3} \rangle$. Uredite ga s števnim urejanjem. Kakšno je stanje v polju C, ko uredimo zaporedje?
4. $I = \langle$Radegost je bog gostoljubnosti.$\rangle$ Zaporedje I uredite s števnim urejanjem.

5. Implementirajte obe različici števnega urejanja (tisto, ki uporablja abecedo za določitev dolžine polja C, in tisto, ki najprej določi interval). Preverite ga na različnih vhodnih zaporedjih I z različnimi abecedami. Ali ima določanje indeksa elementa v abecedi velik vpliv na hitrost izvajanja algoritma?

6. Naj bo $I = \langle 15, 2356, 112, 999, 3 \rangle$. Uredimo ga s korenskim urejanjem.

7. Implemetirajte korensko urejanje za (1) urejanje celih števil, za (2) urejanje nizov in za (3) urejanje števil s plavajočo vejico.

8. Implementirajte urejanje z vedri in ga primerjajte z algoritmom hitrega urejanja. Oba algoritma preverite z naslednjimi podatki:

   - enakomerno porazdeljeni podatki,
   - podatki, porazdeljeni po Gaussovi in Laplaceovi porazdelitvi,
   - podatki, združeni v gruče,
   - monotono naraščajoči in monotono padajoči podatki in
   - podatki z enako vrednostjo.

Poglavje 2

Iskanje vzorca v zaporedju

Zaporedje znakov je najpogostejši način zapisa podatkov, zato je iskanje danega zaporedja znakov v ciljnem zaporedju zelo pogosta naloga v množici aplikacij: pri operacijskih sistemih, zbirkah podatkov, urejevalnikih besedil, spletnih brskalnikih, iskalnikih v bioinformatiki in drugje. Čeprav je naloga na prvi pogled enostavna, je v primeru zelo dolgih zaporedij treba razmišljati tudi o hitrosti delovanja algoritmov. Ločimo med natančnimi in približnimi algoritmi. Približnih si ne bomo ogledali, dober pregled leteh pa najdemo v [13]. Tudi natančne algoritme delimo na dve skupini, tokrat glede na *obstojnost* zaporedja. Če se zaporedje ne spreminja, ga je smiselno predobdelati in umestiti v primerne iskalne strukture (sekljalne tabele, iskalna drevesa, priponska polja ali priponska drevesa). V nasprotnem primeru, ko pričakujemo, da se bo zaporedje spreminjalo, pa poskusimo izkoristiti kombinatorične lastnosti vzorca za morebitno hitrejše premikanje po preiskovanem zaporedju. V tem poglavju bomo spoznali nekaj algoritmov iz druge skupine. Odličen pregled najdemo v [14], kjer je razloženih kar 34 algoritmov.

Naj bo **vzorec** $P = \langle \varsigma_j \rangle$, $0 \leq j < |P|$, katerega ujemanje iščemo v **zaporedju** $I = \langle \sigma_i \rangle$, $0 \leq i < |I|$, $(\varsigma_j, \sigma_i) \in \Sigma$, $m = |P|$, $n = |I|$, $m \leq n$. I preiskujemo z **drsečim oknom**, ki *vidi* le del zaporedja I dolžine m. Simbole iz drsečega okna in simbole iz P nato med seboj primerjamo, čemur pravimo **preizkus**. Po opravljenem preizkusu drseče okno pomaknemo vzdolž zaporedja.

2.1 Naivni pristop

Naivni algoritem (angl. brute force) postavi drseče okno na začetek zaporedja in primerja simbole $\sigma_i \in I$ ter $\varsigma_j \in P$. Če so vsi simboli enaki, algoritem sporoči, da je našel ujemanje, sicer pomakne drseče okno za eno mesto in ponovi postopek. Primer 2.1 kaže nekaj korakov naivnega algoritma, kjer rdeče obarvana simbola opozarjata na prva znaka, ki se ne ujemata.

$$\begin{aligned} I &= \langle \texttt{rokirokyroirokyrokyroyt} \rangle \\ P &= \langle \texttt{rokyroy} \rangle \end{aligned}$$

$$\begin{aligned} I &= \langle \texttt{rokirokyroirokyrokyroyt} \rangle \\ P &= \phantom{\texttt{r}}\langle \texttt{rokyroy} \rangle \end{aligned}$$

$$\begin{aligned} I &= \langle \texttt{rokirokyroirokyrokyroyt} \rangle \\ P &= \phantom{\texttt{ro}}\langle \texttt{rokyroy} \rangle \end{aligned}$$

$$\begin{aligned} I &= \langle \texttt{rokirokyroirokyrokyroyt} \rangle \\ P &= \phantom{\texttt{rok}}\langle \texttt{rokyroy} \rangle \end{aligned}$$

$$\begin{aligned} I &= \langle \texttt{rokirokyroirokyrokyroyt} \rangle \\ P &= \phantom{\texttt{roki}}\langle \texttt{rokyroy} \rangle \end{aligned}$$

$$\begin{aligned} I &= \langle \texttt{rokirokyroirokyrokyroyt} \rangle \\ P &= \phantom{\texttt{rokir}}\langle \texttt{rokyroy} \rangle \end{aligned}$$

Primer 2.1: Nekaj korakov naivnega algoritma

Naivni algoritem ima časovno zahtevnost $O(m \cdot n)$. Da bi jo (morda) zmanjšali, moramo razmisliti, kako bi povečali premike drsečega okna ne da bi kakšno ujemanje spregledali. Algoritmi, ki jih bomo spoznali v nadaljevanju, razen naslednjega, uporabljajo to strategijo. Rabin-Karpov postopek poskuša pohitriti iskanje ujemanja na drugačen način.

2.2 Rabin-Karpov algoritem

Algoritem sta razvila Michael O. Rabin in Richard M. Karp [15]. Ocena najslabše časovne zahtevnosti njunega algoritma se sicer ne razlikuje od naivnega algoritma, zaradi česar ga v praksi redko uporabljamo za iskanje enega vzorca v zaporedju. Zelo uspešen pa je za iskanje več vzorcev hkrati

(angl. multiple pattern matching) [16, 17], različico Rabin-Karpovega algoritma pa pogosto najdemo tudi pri preverjanju plagiatov [18].

Rabin-Karpov algoritem izračunava **zgostitveno funkcijo** (angl. hash function) dela zaporedja I v drsečem oknu in jo primerja z vrednostjo zgostitvene funkcije vzorca P. Prav dobra izbira in predvsem učinkovito računanje zgostitvene funkcije sta ključna prispevka Rabin-Karpovega algoritma.

Zgostitvena funkcija pretvori zaporedje v številko, ki jo imenujemo **zgostitvena vrednost** (angl. hash value). Rabin-Karpov algoritem temelji na dejstvu, da imata dve enaki zaporedji tudi enako zgostitveno vrednost. Na žalost pa odkrijemo dve težavi:

- različna zaporedja lahko imajo enako zgostitveno vrednost, še posebej, če zgostitvena funkcija ni dobro zasnovana, zaradi česar moramo v primeru ujemanja zgostitvenih vrednosti vzorca in dela zaporedja primerjati njune znake;
- izračun zgostitvene funkcije zna biti računsko zahteven.

Rabin-Karp sta za vzorec $P = \langle \varsigma_j \rangle$, $\varsigma_j \in \Sigma$, z dolžino $m = |P|$, predlagala naslednjo **zgostitveno funkcijo**:

$$rk(P_{0,m-1}) = (\varsigma_0 \cdot 2^{m-1} + \varsigma_1 \cdot 2^{m-2} + \cdots + \varsigma_{m-1} \cdot 2^0)\ mod\ q, \tag{2.1}$$

kjer je q veliko število (praviloma je to največja vrednost nepredznačene celoštevilske spremenljivke, to je $2^{32} - 1$ oziroma $2^{64} - 1$).

Izračunajmo vrednost zgostitvene funkcije za vzorec $P = \langle \mathsf{rokyroy} \rangle$, $m = 7$ in $\Sigma = \{\mathsf{i\ k\ o\ r\ y}\}$. Glede na podano abecedo bomo pri izračunu zgostitvene vrednosti nadomestili znak i z vrednostjo 0, znak k z 1, znak o z 2, znak r se bo preslikal v 3, znak y pa v vrednost 4 (glej primer 2.2).

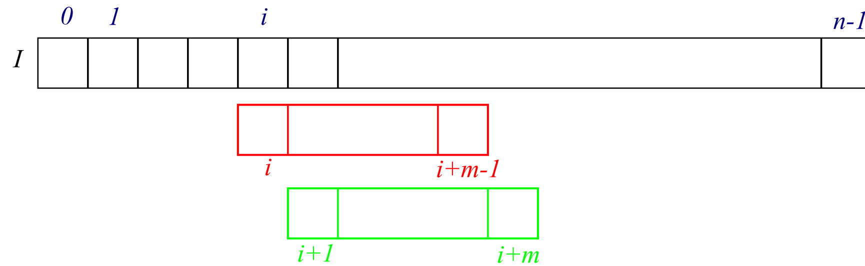
$$rk(P_{0,6}) = 3 \cdot 2^6 + 2 \cdot 2^5 + 1 \cdot 2^4 + 4 \cdot 2^3 + 3 \cdot 2^2 + 2 \cdot 2^1 + 4 \cdot 2^0 = 324$$

Primer 2.2: Izračun zgostitvene funkcije

Ključ učinkovitosti Rabin-Karpovega algoritma leži v računanju zgostitvene funkcije. Algoritem uporablja tako imenovano **vrtečo se zgostitveno funkcijo** (angl. rolling hash function). Za zaporedje I vrednost zgostitvene funkcije v celoti izračunamo samo na začetku, potem pa med premikanjem vzdolž zaporedja uporabimo že izračunano vrednost ter znanje o tem, kako je zgostitvena vrednost izračunana. Vrtečo se zgostitveno funkcijo podaja enačba 2.2.

$$rk(I_{i+1,i+m}) = (2 \cdot (rk(I_{i,i+m-1}) - \sigma_i \cdot 2^{m-1}) + \sigma_{i+m}) \ mod \ q \tag{2.2}$$

Enačbo 2.2 razložimo s sliko 2.1. Predpostavimo, da smo do zdaj izračunali že vse vrednosti zgostitvene funkcije do položaja i, vrednost $rk(I_{i,i+m-1})$ torej poznamo. To vrednost smo izračunali glede na znake, ki so znotraj rdečega drsečega okna. Ko drseče okno pomaknemo za eno mesto (na položaj zelenega okna), vidimo, da znak na položaju i ne nastopa več v izračunu zgostitvene vrednosti, zato moramo ta znak, pomnožen s potenco 2^{m-1}, odšteti. Preostali znaki v zelenem oknu pridobijo na pomembnosti. Pomnožiti bi jih morali z za eno večjo vrednostjo potence 2 (glej enačbo 2.1), kot so bili pomnoženi do zdaj. V primeru vrteče se zgostitvene funkcije pa je dovolj, da to storimo z vrednostjo predhodne funkcije $rk(I_{i,i+m-1})$. Nazadnje prištejemo še vrednost znaka σ_{i+m}. Da preprečimo morebitno prekoračitev, vrednost izračunamo po modulu q.



Slika 2.1: Shematski prikaz vrteče se zgostitvene funkcije

Opravimo še nekaj izračunov za zaporedje iz primera 2.1. Vrednost $rk(P_{0,6}) = 324$ smo že izračunali. Preiskovanje zaporedja I začnemo na položaju $i = 0$ in z enačbo 2.1 dobimo $rk(I_{0,6}) = 289$. Nekaj nadaljnjih izračunov vrteče se zgostitvene funkcije, določene z enačbo 2.2, kaže primer 2.3. Ker je vrednost $rk(I_{15,21}) = rk(P_{0,6})$, primerjamo še posamezne simbole, preden sporočimo, da smo ujemanje vzorca v zaporedju našli.

Rabin-Karpov algoritem je možno implementirati zelo učinkovito. Množenje z 2 nadomestimo z logičnim pomikom v desno, operacijo modula pa opravimo kar z deklaracijo spremenljivke, v katero shranimo, kot smo že omenili, vrednost rk.

$$rk(I_{1,7}) = 2(289 - 3 \cdot 2^6) + 4 = 198$$
$$rk(I_{2,8}) = 2(198 - 2 \cdot 2^6) + 3 = 143$$
$$rk(I_{3,9}) = 2(143 - 1 \cdot 2^6) + 2 = 160$$
$$rk(I_{4,10}) = 2(160 - 0 \cdot 2^6) + 0 = 320$$
$$rk(I_{5,11}) = 2(320 - 3 \cdot 2^6) + 3 = 259$$
$$rk(I_{6,12}) = 2(259 - 2 \cdot 2^6) + 2 = 264$$
$$rk(I_{7,13}) = 2(264 - 1 \cdot 2^6) + 1 = 401$$
$$rk(I_{8,14}) = 2(401 - 4 \cdot 2^6) + 4 = 294$$
$$rk(I_{9,15}) = 2(294 - 3 \cdot 2^6) + 3 = 207$$
$$rk(I_{10,16}) = 2(207 - 2 \cdot 2^6) + 2 = 160$$
$$rk(I_{11,17}) = 2(160 - 0 \cdot 2^6) + 1 = 321$$
$$rk(I_{12,18}) = 2(321 - 3 \cdot 2^6) + 4 = 262$$
$$rk(I_{13,19}) = 2(262 - 2 \cdot 2^6) + 3 = 271$$
$$rk(I_{14,20}) = 2(271 - 1 \cdot 2^6) + 2 = 416$$
$$rk(I_{15,21}) = 2(416 - 4 \cdot 2^6) + 4 = 324$$

Primer 2.3: Izračuni vrteče se zgostitvene funkcije

2.3 Knut-Morris-Prattov algoritem

Algoritem ima zanimivo zgodovino. Zasnoval ga je James H. Morris, nekoliko pozneje pa ga je neodvisno izumil tudi D. Knuth. Morris in Pratt sta leta 1970 objavila tehniško poročilo z opisom algoritma, 7 let pozneje pa so algoritem objavili vsi trije skupaj [19]. Algoritem Knut-Moris-Pratt (algoritem KMP) deluje v dveh korakih:

- pred začetkom preiskovanja zaporedja I preveri lastnosti vzorca P;
- glede na lastnosti vzorca v vsaki iteraciji določi, za koliko mest je možno premakniti drseče okno tako, da P nikakor ne bo spregledan.

Idejo algoritma KMP razložimo s primerom 2.4.

Algoritem hkrati premika indeksa i in j do prvega neujemanja med σ_i in ς_j. V našem primeru se to zgodi pri četrti primerjavi, ko sta $i = j = 3$ in

```
i: 01234567890123456789012
I=⟨rokirokyroirokyrokyroyt⟩
P=⟨rokyroy⟩
j: 0123456
```

Primer 2.4: Ideja algoritma KMP – začetno stanje

primerjamo $\sigma_3 = \mathtt{i}$ in $\varsigma_3 = \mathtt{y}$. Ker so znaki $\sigma_1, \sigma_2, \sigma_3 \neq \varsigma_0$, lahko pomaknemo drseče okno za štiri mesta. Situacijo kaže primer 2.5.

```
i: 01234567890123456789012
I=⟨rokirokyroirokyrokyroyt⟩
P=    ⟨rokyroy⟩
j:     0123456
```

Primer 2.5: Ideja algoritma KMP – prvi premik

Ujemanje zaporedja in vzorca je skoraj popolno, razen na mestu $j = 6$ ($i = 10$). S premikom moramo biti tokrat pazljivejši. Pri primerjavah znakov v zaporedju in vzorcu smo prešli tudi črki r in o na položajih $j = 4$ in $j = 5$, ki se ujemata z začetkom vzorca. Ta znaka bi lahko pomenila začetek iskanega vzorca, zato moramo premik drsečega okna zmanjšati (glej primer 2.6).

```
i: 01234567890123456789012
I=⟨rokirokyroirokyrokyroyt⟩
P=        ⟨rokyroy⟩
j:         0123456
```

Primer 2.6: Ideja algoritma KMP – drugi premik

Za prva znaka iz I in P vemo, da se ujemata, zato postavimo $j = 2$ in $i = 10$. Poskus se takoj zaključi, saj je $\varsigma_2 \neq \sigma_{10}$. Opravimo premik drsečega okna za štiri mesta, kot kaže primer 2.7. Nastopi podoben primer

```
i: 01234567890123456789012
I=⟨rokirokyroirokyrokyroyt⟩
P=           ⟨rokyroy⟩
j:            0123456
```

Primer 2.7: Ideja algoritma KMP – tretji premik

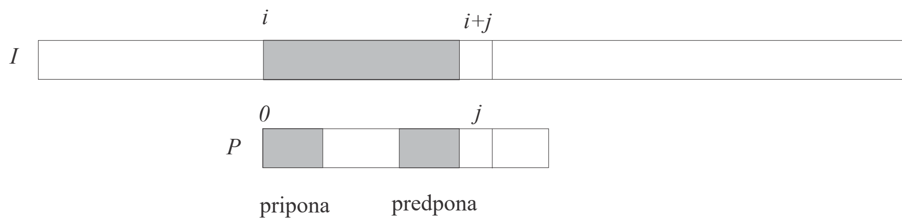
kot prej. Neujemanje smo zaznali pri zadnjem znaku vzorca, zato moramo

premik zmanjšati; premaknemo ga za štiri mesta. Ujemanje tokrat najdemo (primer 2.8).

```
i: 01234567890123456789012
I=⟨rokirokyroirokyrokyroyt⟩
P=                ⟨rokyroy⟩
j:                 0123456
```

Primer 2.8: Ideja algoritma KMP – četrti premik

Za določanje dolžine premika drsečega okna algoritem KMP **predobdela** P. Predpostavimo, da je neujemanje povzročil znak na položaju j, $0 < j < m$. Preveriti moramo, ali se znaki neposredno pred j ujemajo z začetkom vzorca. Znakom na začetku vzorca bomo rekli **pripona** (angl. suffix), neposredno pred znakom ς_j pa **predpona** (angl. prefix) (glej sliko 2.2). Dolžina ujemanja pripone in predpone določa, za koliko moramo zmanjšati premik drsečega okna, če je povzročil neujemanje znak ς_j. Če ujemanja pripone in predpone ni, bi lahko premaknili okno za $j + 1$ mest (kot bomo videli pozneje, algoritem KMP opravi premik samo za j mest).



Slika 2.2: Pripona in predpona pri algoritmu KMP

Rezultat analize vzorca zapišemo v razpredelnico 2.1, kjer vrstica $kmpNext$ določa dolžino ujemanja pripone in predpone. Poglejmo, kako jo sestavimo v primeru vzorca $P = \langle$rokyroy$\rangle$.

- Prvi element vedno postavimo na vrednost -1, saj pripona in predpona ne obstajata ($kmpNext_0 = -1$), vrednost -1 pa bo, kot bomo videli, omogočala premik drsečega okna.
- Pri $j = 1$ pripona in predpona predstavljata isti znak, zato tudi tukaj ne moremo ugotavljati ujemanja pripone in predpone; postavimo $kmpNext_1 = 0$.
- Pri $j = 2$ je predpona $P_0 =$ r in pripona $P_1 =$ o. Ujemanja ni, zato postavimo $kmpNext_2 = 0$.

- Enako kot v primeru $j = 2$, se tudi na položajih $j = 3$ in $j = 4$ priponi in predponi ne ujemata, zato postavimo $kmpNext_3 = 0$ in $kmpNext_4 = 0$.
- Pri $j = 5$ zaznamo ujemanje med pripono in predpono v dolžini enega znaka, $P_0 = P_4 =$ r, zato $kmpNext_5 = 1$.
- Ko je $j = 6$, se pripona in predpona ujemata v dolžini dveh znakov, zato je $kmpNext_6 = 2$.

Razpredelnica 2.1: Obdelava vzorca po metodi KMP

j	0	1	2	3	4	5	6
P	r	o	k	y	r	o	y
$kmpNext$	-1	0	0	0	0	1	2

Ko smo predobdelali vzorec P in napolnili *kmpNext*, določimo dolžino premika s z enačbo 2.3, kjer je j mesto v P, kjer je prišlo do neujemanja.

$$s = j - kmpNext_j \tag{2.3}$$

Ko najdemo ujemanje, opravimo premik za $m + 1$ znakov. Če pa bi želeli najti tudi morebitne pojavitve ujemanja vzorca, ki se začnejo znotraj drsečega okna, se premaknemo samo za en znak. Korake algoritma KMP podaja primer 2.9.

```
I = ⟨rokirokyroirokyrokyroyt⟩
P = ⟨rokyroy⟩                    s = j − kmpNext_j = 3 − 0 = 3
P =    ⟨rokyroy⟩                 s = j − kmpNext_j = 0 − (−1) = 1
P =     ⟨rokyroy⟩                s = j − kmpNext_j = 6 − 2 = 4
P =         ⟨rokyroy⟩            s = j − kmpNext_j = 2 − 0 = 2
P =           ⟨rokyroy⟩          s = j − kmpNext_j = 0 − (−1) = 1
P =            ⟨rokyroy⟩         s = j − kmpNext_j = 6 − 2 = 4
P =                ⟨rokyroy⟩     ujemanje; s = 7
P =                       ⟨rokyroy⟩
```

Primer 2.9: Delovanje algoritma KMP

2.4 Horspoolov algoritem

Horspoolov algoritem [20] je poenostavitev algoritma Boyer-Moore [21], ki pa si ga zaradi zapletenosti ne bomo ogledali. Horspoolov algoritem preis-

kuje vzorec od desne proti levi, zato v tej smeri tudi predobdelamo vzorec. Za vse znake $\sigma_i \in \Sigma$, ki tvorijo sporočilo $I = \langle \sigma_i \rangle$, določimo vrednost hevristike na naslednji način:

- Če zadnji znak vzorca P_{m-1} ne obstaja nikjer drugje v P, postavimo odmik na m.
- Če v P obstaja več enakih znakov, postavimo vrednost odmika na prvi znak z desne. Če je med enakimi znaki tudi zadnji znak v P, tega ne upoštevamo.
- Če znak v P ne obstaja, je njegov odmik m.

Naj bo $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\}$ in $P = \langle \mathsf{bdbda} \rangle$. Zadnji znak vzorca a se ne pojavi nikjer drugje v P, prav tako $\mathsf{c} \notin P$. Razpredelnica 2.2 kaže odmike za Horspoolov algoritem.

Razpredelnica 2.2: Hevristika Horspoolovega algoritma, ko se zadnji znak ne ponovi nikjer drugje v P

σ_i	a	b	c	d
Horspool(σ_i)	5	2	5	1

Horspoolov algoritem za določanje dolžine premika ne uporablja znaka σ_i, ki je povzročil neujemanje, ampak vedno **skrajno desni znak** v drsečem oknu zaporedja I. Premikanje skozi I s Horpoolovim algoritmom predstavi primer 2.10, premike pa nadziramo z odmiki, določenimi v razpredelnici 2.3.

Razpredelnica 2.3: Hevristika Horspoolovega algoritma, ko se zadnji znak še pojavi v vzorcu

σ_i	A	C	G	T
Horspool(σ_i)	1	6	2	8

```
I = ⟨GCATCGCAGAGAGTATACAGTACG⟩
P = ⟨GCAGAGAG⟩                    premik za 1 znak

I = ⟨GCATCGCAGAGAGTATACAGTACG⟩
P =  ⟨GCAGAGAG⟩                   premik za 2 znaka

I = ⟨GCATCGCAGAGAGTATACAGTACG⟩
P =   ⟨GCAGAGAG⟩                  premik za 2 znaka

I = ⟨GCATCGCAGAGAGTATACAGTACG⟩
P =     ⟨GCAGAGAG⟩                premik za 2 znaka

I = ⟨GCATCGCAGAGAGTATACAGTACG⟩
P =       ⟨GCAGAGAG⟩              premik za 1 znak

I = ⟨GCATCGCAGAGAGTATACAGTACG⟩
P =        ⟨GCAGAGAG⟩             premik za 8 znakov

I = ⟨GCATCGCAGAGAGTATACAGTACG⟩
P =                ⟨GCAGAGAG⟩     premik za 2 znaka
```

Primer 2.10: Premiki Horspoolovega algoritma, kjer rdeč simbol določa dolžino premika.

2.5 Sundayev algoritem

Sunday [22] je ugotovil, da je možno za določitev premika vzorca uporabiti prvi naslednji znak izven drsečega okna. Predobdelava P je podobna kot pri Horspoolovem algoritmu, le da upoštevamo, da se bomo premikali glede na naslednji simbol v zaporedju.

Tudi Sundayev algoritem razložimo s primerom. Razpredelnica 2.4 podaja predobdelavo vzorca, primer 2.11 pa premikanje skozi zaporedje, kjer je rdeče označena črka kazalec v razpredelnico premikov.

Razpredelnica 2.4: Hevristika Sundayevega algoritma

σ_i	A	C	G	T
Sunday(σ_i)	2	7	1	9

Pri Horspoolovem in Sundayevem algoritmu je dejansko vseeno, s katere strani začnemo preiskovati vzorec. Sunday je to dejstvo izkoristil tako, da je

```
I = ⟨GCATCGCAGAGAGTATACAGTACG⟩
P = ⟨GCAGAGAG⟩                    premik za 1 znak

I = ⟨GCATCGCAGAGAGTATACAGTACG⟩
P =  ⟨GCAGAGAG⟩                   premik za 2 znaka

I = ⟨GCATCGCAGAGAGTATACAGTACG⟩
P =    ⟨GCAGAGAG⟩                 premik za 2 znaka

I = ⟨GCATCGCAGAGAGTATACAGTACG⟩
P =      ⟨GCAGAGAG⟩               premik za 9 znakov

I = ⟨GCATCGCAGAGAGTATACAGTACG⟩
P =               ⟨GCAGAGAG⟩ premik za 7 znak

I = ⟨GCATCGCAGAGAGTATACAGTACG⟩
P =                      ⟨GCAGAGAG⟩
```

Primer 2.11: Premiki Sundayevega algoritma

najprej preveril tiste znake v vzorcu, ki se v zaporedju najredkeje pojavijo, seveda, če to informacijo poznamo. V našem primeru se znak C pojavlja v I najredkeje, zato bi ga v vzorcu preverili najprej. Število primerjav bi tako dodatno zmanjšali, kar lahko bralec preveri sam.

Naloge

1. Razložite Rabin-Karpovo zgoščevalno funkcijo.

2. Realizirajte naivno metodo in Rabin-Karpov algoritem ter preverite, ali je Rabin-Karpov algoritem hitrejši od naivne metode. Kako dolžina vzorca vpliva na učinkovitost obeh algoritmov?

3. Naj bo $P = \langle$dedecinjedec$\rangle$. Sestavite razpredelnice premikov za algoritme KMP, Horspool in Sunday.

4. Naj bo sporočilo $I = \langle$Roža perunika je dobila ime po perunu.$\rangle$ in vzorec $P = \langle$perun$\rangle$. V koliko korakih se zaključijo algoritmi KMP, Horspool in Sunday?

5. V prejšnji nalogi izboljšajte Sundayev algoritem z upoštevanjem pogostosti pojavljanja znakov.

6. Implementirajte algoritme KMP, Horspool in Sunday.

7. V literaturi (na primer [14]) ali na spletu poiščite razlago še kakšnega algoritma za iskanje vzorcev v zaporedjih, implementirajte ga in primerjajte z obravnavanimi algoritmi.

Poglavje 3

Najkrajša razdalja urejanja zaporedij

Poleg tega, da hitro najdemo dele zaporedja, ki se ujemajo z vzorcem, je lahko koristna informacija tudi, koliko operacij urejanja potrebujemo, da bi iz zaporedja I prešli v zaporedje J. Operacije urejanja so **vrinjanje** (angl. insertion), **brisanje** (angl. deletion) in **prepisovanje** (angl. overwriting) znakov. To nalogo izvajajo na primer črkovalniki, ki nam poskušajo ponuditi najbližjo besedo tisti, ki smo jo napisali in jo črkovalnik ne najde v svojem slovarju, ali spletni brskalniki, ki želijo uganiti pravo besedo, ki smo jo podali, a zadetkov ni bilo. V razpredelnici 3.1 je nekaj primerov, kako preiti iz zaporedja I v zaporedje J, kjer smo rdeče znake prepisali, zelene vrinili, tiste, ki smo jih izbrisali, pa prečrtali. Številka v zadnjem stolpcu nam pove število operacij urejanja; imenovali jo bomo tudi **cena urejanja** ali **razdalja urejanja zaporedij**.

Razpredelnica 3.1: Razdalja urejanja med zaporedjema I in J

I	J	$I \to J$	cena urejanja
drevo	predivo	predivo	3
vodovod	dovod	~~vo~~dovod	2
gretje	petletje	~~g~~petletje	5

Število operacij urejanj med zaporedjema I in J je lahko različno. Na primer zaporedje $I = \langle \text{drevo} \rangle$ lahko najprej zbrišemo s petimi operacijami, nato pa napišemo/vrinemo 7 znakov zaporedja $J = \langle \text{predivo} \rangle$, skupaj torej 12 operacij urejanja. Zagotovo pa se je smiselno osredotočiti na najmanjše število operacij urejanj, ki ji pravimo tudi **Levenštejnova razdalja**, (defi-

niral jo je ruski matematik Vladimir Iosifovič Levenštejn).

Naj bosta zaporedji $I = \langle\sigma_i\rangle$, $0 \le i < n$, in $J = \langle\varsigma_j\rangle$, $0 \le j < m$, $(\sigma_i, \varsigma_j) \in \Sigma$. Iz zaporedij I in J v splošnem ne znamo takoj določiti najkrajše razdalje urejanja, zato problem zmanjšujemo tako dolgo, da dobimo enostavno rešljive **elementarne probleme**. Rešitve elementarnih problemov nam potem pomagajo rešiti nekoliko večje probleme, ti spet večje, dokler ne rešimo zadanega problema. Strategiji pravimo **dinamično programiranje** [1] in jo boste bolje spoznali pri drugih predmetih.

V razpredelnici 3.2 vidimo tri možne elementarne probleme za našo nalogo iskanja najkrajše razdalje urejanja zaporedij. Pri prvem primeru primerjamo znak $\sigma_i \in I$ s praznim zaporedjem. Če želimo preiti iz znaka σ_i v

Razpredelnica 3.2: Elementarni problemi najkrajše razdalje urejanja

znak zaporedja I	σ_i	$-$	σ_i
znak zaporedja J	$-$	ς_j	ς_j
razdalja urejanja	1	1	d

prazno zaporedje, moramo uporabiti eno operacijo urejanja (brisanje), kar nam da razdaljo urejanja 1. Druga situacija je analogna. Tokrat primerjamo $\varsigma_j \in J$ s praznim zaporedjem I in spet je razdalja urejanja 1. V zadnjem primeru imamo na položajih i in j znaka obeh zaporedij. Razdalja urejanja je odvisna od tega, ali sta σ_i in ς_j enaka ali različna. V prvem primeru je seveda dolžina urejanja 0, v drugem pa 1, kar zapišemo kot

$$d_{i,j} = \begin{cases} 0, & \sigma_i = \varsigma_j \\ 1, & \text{sicer.} \end{cases} \tag{3.1}$$

Problem začnemo reševati nad celotnima nizoma I in J. Predpostavimo, da bo za nas ta problem rešila neka **optimizacijska funkcija** $E_{n-1,m-1}$, kjer parametra n in m predstavljata dolžini zaporedij I in J, kot smo že definirali. Če je ena (ali celo obe) od vrednosti n in m enaka 0, je problem elementaren in trivialno rešljiv, sicer pa ga moramo postopoma zmanjševati, dokler ne pridemo do elementarnih problemov. Zmanjševanje velikosti problema bomo realizirali s postopnim zmanjševanjem indeksov i in j, $n > i \ge 0$, $m > j \ge 0$. Optimizacijska funkcija bo tako v nekem trenutku iskala najmanjše število operacij urejanja le za dele nizov, in sicer za $I_{0,i}$ in $J_{0,j}$, kar bomo zapisali kot $E_{i,j}$. Kot smo pokazali (glej razpredelnico 3.2), obstajajo tri možnosti, ki jih mora na vsaki ravni reševanja problema obravnavati funkcija $E_{i,j}$. Med njimi mora izbrati tistega, ki da najkrajšo razdaljo urejanja, kar formalno zapišemo z enačbo 3.2 (pravimo ji tudi Bellmanova enačba).

$$E_{i,j} \quad = \quad \min\{1 + E_{i-1,j},\ 1 + E_{i,j-1},\ d_{i,j} + E_{i-1,j-1}\} \tag{3.2}$$

Enačba 3.2 je rekurzivna, vsakega izmed treh podproblemov zmanjšuje do enostavne rešitve. Rešitev celotnega problema dobimo, ko se funkcija vrne iz rekurzije. Delovanje funkcije E si osvetlimo s primerom 3.1. Ker je $n = |I| = 2$ in $m = |J| = 4$, moramo poiskati rešitev optimizacijske funkcije $E_{n-1,m-1} = E_{1,3}$, s čimer sprožimo rekurzivni izračun, prikazan s primerom 3.2. Vrednost indeksov $i = -1$ oziroma $j = -1$ označuje prazno zaporedje. Vidimo, da nam enačba 3.2 vrne pravi rezultat; najkrajša razdalja urejanja zaporedij $\langle$ba$\rangle$ in $\langle$goba$\rangle$ je 2.

```
i:    01
I = ⟨ba⟩
J = ⟨goba⟩
j:    0123
```

Primer 3.1: Zaporedji I in J za določitev najkrajše razdalje urejanja

$$
\begin{aligned}
E_{1,3} &= min\{1+E_{0,3}, \quad 1+E_{1,2}, \quad d_{1,3}+E_{0,2}\}\\
E_{0,3} &= min\{1+E_{-1,3}, 1+E_{0,2}, \quad d_{0,3}+E_{-1,2}\}\\
E_{1,2} &= min\{1+E_{0,2}, \quad 1+E_{1,1}, \quad d_{1,2}+E_{0,1}\}\\
E_{0,2} &= min\{1+E_{-1,2}, 1+E_{0,1}, \quad d_{0,2}+E_{-1,1}\}\\
E_{1,1} &= min\{1+E_{0,1}, \quad 1+E_{1,0}, \quad d_{1,1}+E_{0,0}\}\\
E_{0,1} &= min\{1+E_{-1,1}, 1+E_{0,0}, \quad d_{0,1}+E_{-1,0}\}\\
E_{1,0} &= min\{1+E_{0,0}, \quad 1+E_{1,-1}, d_{1,0}+E_{0,-1}\}\\
E_{0,0} &= min\{1+E_{-1,0}, 1+E_{0,-1}, d_{0,0}+E_{-1,-1}\}
\end{aligned}
$$

$$
\begin{aligned}
E_{0,0} &= min\{1+1, \qquad 1+1, \qquad 1+0\} = 1;\\
E_{1,0} &= min\{1+1, \qquad 1+2, \qquad 1+1\} = 2;\\
E_{0,1} &= min\{1+2, \qquad 1+1, \qquad 1+1\} = 2;\\
E_{1,1} &= min\{1+2, \qquad 1+2, \qquad 1+1\} = 2;\\
E_{0,2} &= min\{1+3, \qquad 1+2, \qquad 0+2\} = 2;\\
E_{1,2} &= min\{1+2, \qquad 1+2, \qquad 1+2\} = 3;\\
E_{0,3} &= min\{1+4, \qquad 1+2, \qquad 1+3\} = 3;\\
E_{1,3} &= min\{1+3, \qquad 1+3, \qquad 0+2\} = 2;
\end{aligned}
$$

Primer 3.2: Rekurzivni izračun najkrajše razdalje urejanja zaporedij $I = \langle \mathsf{ba} \rangle$ in $J = \langle \mathsf{goba} \rangle$

3.1 Wagner-Fischerjev algoritem

Problem v praksi rešimo z Wagner-Fischerjevim algoritmom[1] [23]. Algoritem uporablja matriko, katere začetno stanje za zaporedji iz primera 3.3 vidimo v razpredelnici 3.3.

$$
\begin{aligned}
I &= \langle \texttt{predavanja} \rangle\\
J &= \langle \texttt{sprehajanje} \rangle
\end{aligned}
$$

Primer 3.3: Zaporedji I in J za določitev najkrajše razdalje ujemanja

[1] Kot je zapisal Navarro [13], je izumiteljev tega algoritma več, najpogosteje pa se imenuje po Wagnerju in Fischerju.

Razpredelnica 3.3: Inicializacija Wagner-Fischerjeve matrike

$W =$

i		-1	0	1	2	3	4	5	6	7	8	9
j			p	r	e	d	a	v	a	n	j	a
-1		0	1	2	3	4	5	6	7	8	9	10
0	s	1										
1	p	2										
2	r	3										
3	e	4										
4	h	5										
5	a	6										
6	j	7										
7	a	8										
8	n	9										
9	j	10										
10	e	11										

Vrednosti so v prvem stolpcu in prvi vrstici že določene. Te vrednosti nam povedo, kakšna je najkrajša razdalja urejanja zaporedij, če je eno od zaporedij prazno. Za preostale elemente matrike W uporabimo enačbo 3.2, ki jo tokrat zapišemo kot

$$W_{i,j} = \min\{1 + W_{i-1,j},\ 1 + W_{i,j-1},\ d_{i,j} + W_{i-1,j-1}\}. \qquad (3.3)$$

Postopek polnjenja W je očitno preprost. Matriko polnimo s sprehodom od leve proti desni od zgoraj navzdol. Na položaju i, j vzamemo vrednosti neposrednih sosedov $W_{i-1,j}$ in $W_{i,j-1}$ ter jima prištejemo 1. Tudi elementu $W_{i-1,j-1}$ prištejemo 1, če sta elementa $I_i \neq J_j$, sicer ne prištejemo ničesar. Najkrajše število urejanja zaporedij dobimo v skrajno desnem spodnjem elementu matrike W. Napolnjeno matriko W za primer 3.3 kaže razpredelnica 3.4. Implementacija algoritma je enostavna, problem pa rešimo v času $O(n\ m)$, torej v kvadratnem času.

Naloge

1. Poščite najkrajšo razdaljo urejanja zaporedij $I = \langle \mathsf{Mura} \rangle$ in $J = \langle \mathsf{ura} \rangle$ z rekurzivnim izračunom.
2. Z Wagner-Fischerjevim algoritmom poiščite najkrajšo razdaljo urejanja zaporedij $I = \langle \mathsf{Prekmurje} \rangle$ in $J = \langle \mathsf{Primorje} \rangle$ ter $I =$

Razpredelnica 3.4: Napolnjena Wagner-Fischerjeva matrika W

$W =$

i		−1	0	1	2	3	4	5	6	7	8	9
j			p	r	e	d	a	v	a	n	j	a
−1		0	1	2	3	4	5	6	7	8	9	10
0	s	1	1	2	3	4	5	6	7	8	9	10
1	p	2	1	2	3	4	5	6	7	8	9	10
2	r	3	2	1	2	3	4	5	6	7	8	9
3	e	4	3	2	1	2	3	4	5	6	7	8
4	h	5	4	3	2	2	3	4	5	6	7	8
5	a	6	5	4	3	3	2	3	4	5	6	7
6	j	7	6	5	4	4	3	3	4	5	5	6
7	a	8	7	6	5	5	4	4	3	4	5	5
8	n	9	8	7	6	6	5	5	4	3	4	5
9	j	10	9	8	7	7	6	6	5	4	3	4
10	e	11	10	9	8	8	7	7	6	5	4	**4**

$\langle$znanstvenoraziskovalni$\rangle$ in $J = \langle$analognodigitalni$\rangle$.

3. Implementirajte Wagner-Fischerjev algoritem in rekurzivni algoritem. Z meritvami časa CPU določite njuno časovno zahtevnost. Naj bo dolžina zaporedij $|I| = |J|$. Kakšno dolžino zaporedij zmoreta algoritma v desetinki sekunde?

4. Wagner-Fischerjev algoritem in rekurzivni algoritem nam ne povesta, kaj moramo storiti, da bomo iz zaporedja I prešli v zaporedje J z najmanjšim številom operacij urejanja. Napišite algoritem, ki bo uporabniku tudi pokazal, kako uporabiti operacije urejanja za prehod iz I v J, pri čemer uporabite barvno kodiranje, kot smo ga uporabili v razpredelnici 3.1.

Poglavje 4

Preprosti algoritmi šifriranja

Šifriranje/kriptografija (angl. criptography) je danes znanstvena disciplina z močnim prepletanjem teorije števil in računalništva, v preteklosti pa so bili postopki šifriranja popolnoma algoritmični. Moč takšnih postopkov je, gledano z današnjimi očmi, šibka, a za aplikacije, za katere lahko predvidevamo, da se napadov nanje ne bodo lotili ravno vrhunski strokovnjaki, so še danes dovolj zahtevni. Poleg tega je njihovo programiranje enostavno, zato se je vredno pozabavati tudi z njimi. Nekaj takšnih idej bomo spoznali v nadaljevanju. Povzemamo jih iz uvodnega poglavja knjige avtorjev Trappeja in Washingtona [24]. Aktualne algoritme, na primer algoritme DES, AES in RSA, boste spoznali pri drugem predmetu.

4.1 Pomikalni šifrirnik

Naj bo vhodno sporočilo $I = \langle \sigma_i \rangle$ in Σ abeceda, iz katere sporočilo sestoji, $\sigma_i \in \Sigma$. Najstarejši znani šifrirni sistem naj bi izumil Julij Cezar [24]. Njegovo idejo razložimo s primerom. Zapišimo slovensko abecedo z malimi tiskanimi črkami Σ_s, razširjeno s presledkom, $\Sigma = \Sigma_s \cup \llcorner\!\lrcorner$, in sporočilo I iz primera 4.1.

$$I \;= \langle \texttt{srečamo se pri studencu} \rangle$$

Primer 4.1: Sporočilo I, ki ga bomo šifrirali s pomikalnim šifrirnikom

V postopku šifriranja najprej izpustimo presledke med besedami, saj bi z njihovo pomočjo nepovabljen bralec precej lažje razvozlal sporočilo. Tako naša aktualna abeceda postane samo slovenska abeceda Σ_s (razpredelnica 4.1).

Razpredelnica 4.1: Slovenska abeceda Σ_s

a	b	c	č	d	e	f	g	h	i	j	k	l	m	n	o	p	r	s	š	t	u	v	z	ž
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24

Vsako črko $\sigma_i \in I$ premaknemo za K mest v desno (K imenujemo **ključ**, (angl. key), $K < |\Sigma|$) glede na abecedni vrstni red. Če zmanjka črk na desni, se premaknemo na začetek abecede in nadaljujemo s štetjem. Rezultat šifriranja je zaporedje $O = \langle \varsigma_i \rangle$, $\varsigma_i \in \Sigma_s$, $|O| = |I|$. Šifrirano sporočilo iz primera 4.1 pri $K = 10$ kaže primer 4.2, opisanemu postopku pa pravimo **pomikalni šifrirnik** (angl. shift cipher).

$$O = \langle \texttt{čcomjzačobcščefnožlf} \rangle$$

Primer 4.2: Šifrirano sporočilo s pomikalnim šifrirnikom, ko je $K = 10$

Dešifriranje je enostavno; če poznamo vrednost ključa, črke ς_i šifriranega sporočila premaknemo za K mest v levo glede na položaj v abecedi, ki jo moramo tudi poznati. Nazadnje z logičnim sklepanjem vstavimo še prazna mesta.

Če bi želeli pomikalni šifrirnik sprogramirati, moramo postopek tudi formalizirati, kar storimo z enačbo 4.1, kjer $|\sigma_i|$ in $|\varsigma_i|$ označujeta zaporedno mesto črke v Σ.

$$\begin{aligned} O_i =& (|\sigma_i| + K) \; mod \; |\Sigma| \\ I_i =& (|\varsigma_i| - K) \; mod \; |\Sigma| \end{aligned} \tag{4.1}$$

Čeprav je šifra v primeru 4.2 videti zelo zapletena, pa jo je zelo lahko razvozlati. Dovolj je, da je sporočilo nekoliko daljše in da vemo, v katerem naravnem jeziku je napisan I. Za ta jezik nato uporabimo tabelo verjetnosti pojavljanja posamezne črke v sporočilu. Najpogostejša črka v slovenščini je e, ki se pojavlja z verjetnostjo 0,10707, najredkejša pa f z verjetnostjo 0,0011 [25, 26]. S pazljivo sestave besedila se lahko statistična verjetnost tudi zaobide.

Zakaj pa je šifro, tvorjeno s pomikalnim šifrirnikom, tako lahko dekodirati? Hitro najdemo pravi odgovor. Največja slabost pomikalnih šifrirnikov je, da se črka σ_i vedno preslika v isto črko ς_j. Takšnim šifrirnikom pravimo **monoalfabetni/enočrkovni šifrirniki** (angl. monoalphabetic). Moč šifrirnika bi močno izboljšali in izničili statistične lastnosti naravnega jezika, če to ne bi veljalo; torej, če bi se neka črka σ_i preslikala v različne črke ς_j. Takšni šifrirniki seveda obstajajo, pravimo jim **polialfabetni/veččrkovni** (angl. polyalphabetic) šifrirniki. Prvi veččrkovni šifrirnik je izumil Vigenér.

4.2 Vigenérjev šifrirnik

Vigenérjev šifrirnik uporablja ključ $K = \langle \sigma_k \rangle$, ki, tako kot sporočilo, vsebuje znake iz abecede $\sigma_k \in \Sigma$. Tudi tokrat razložimo postopek šifriranja z zaporedjem I iz primera 4.1, ključ pa naj bo $K = \langle jarilo \rangle$.

Tokrat simbolom sporočila $\sigma_i \in I$ priredimo števila $|\sigma_i|$, ki ustrezajo mestu črke v abecedi (uporabimo slovensko abecedo Σ_s iz razpredelnice 4.1), enako storimo tudi za črke iz ključa (glej primer 4.3).

$I = \langle$	s	r	e	č	a	m	o	s	e	p	r	i	s	t	u	d	e	n	c	u$\rangle$
	18	17	5	3	0	13	15	18	5	16	17	9	18	20	21	4	5	14	2	21

$K = \langle$	j	a	r	i	l	o$\rangle$
	10	0	17	9	12	15

Primer 4.3: Črke sporočila in ključa zamenjamo s števili abecednega vrstnega reda

Števila nato seštejemo po modulu $|\Sigma|$. Ker je najpogosteje dolžina ključa krajša, kot je dolžina sporočila ($|K| < |I|$), števila, ki smo jih dobili iz ključa, ponovimo tolikokrat, da lahko zašifriramo vse znake sporočila, kot kaže primer 4.4.

18	17	5	3	0	13	15	18	5	16	17	9	18	20	21	4	5	14	2	21
10	0	17	9	12	15	10	0	17	9	12	15	10	0	17	9	12	15	10	0
3	17	22	12	12	3	0	18	22	0	4	24	3	20	13	13	17	4	12	21

Primer 4.4: Števila seštejemo po modulu 25

Nazadnje števila pretvorimo v zaporedje znakov iz Σ_s, ki tvorijo šifrirano sporočilo O (glej primer 4.5). Isti znaki vhodnega sporočila se praviloma preslikajo v različne znake šifriranega sporočila.

$O = \langle$`črvllčasvadžčtmmrdlu`$\rangle$

Primer 4.5: Šifrirano sporočilo z Vigenérjevo metodo

Postopek dešifriranja je enak, le da števila, ki jih priredimo črkam šifriranega sporočila O, odštejemo od števil iz ključa K, po modulu $|\Sigma|$. Formalno postopek kodiranja in dekodiranja opišemo z enačbo 4.2.

$$\begin{aligned} O_i =& (|\sigma_i| + |K_{(i \ mod \ |K|)}|) \ mod \ |\Sigma| \\ I_i =& (|\varsigma_i| - |K_{(i \ mod \ |K|)}|) \ mod \ |\Sigma| \end{aligned} \tag{4.2}$$

Če ključa ne poznamo, se hitro znajdemo pred zelo zahtevno nalogo. Število možnih ključev je zavidanja vrednih $|\Sigma|^{|K|}$ (v našem primeru $25^6 = 244.140.625$), kar onemogoča dešifiranje *na roko*, računalniški program pa bo pri krajših dolžinah ključa kljub temu zelo hitro našel rešitev.

4.2.1 Šifrirnik Playfair

Šifrirnik Playfair je med prvo svetovno vojno uporabljala angleška vojska. Šifrirnik je izumil sir Charles Wheatstone, poimenoval pa ga je po svojem prijatelju baronu Playfairu, ki je uspel prepričati angleško vlado, da je šifrirnik začela tudi uporabljati. S pomočjo današnjih računalnikov pa sporočila, šifrirana s tem šifrirnikom, razkrijemo v kratkem času.

Šifrirnik uporablja poljubno zaporedje za ključ, s pomočjo katerega sestavimo šifrirno tabelo. Poglejmo si primer: naj bo ključ $K = \langle \mathsf{playfair} \rangle$. Ponavljajoče znake in morebitne presledke odstranimo; v našem primeru dobimo $K = \langle \mathsf{playfir} \rangle$. Preurejen ključ uvrstimo v šifrirno matriko velikosti 5×5, ključu pa sledijo preostale črke abecede Σ. Ker ima angleška abeceda 26 črk, obravnavamo i in j kot eno črko; kot črko i. Šifrirno matriko χ_{pf} kaže primer 4.6.

$$\chi_{pf} = \begin{matrix} \mathsf{p} & \mathsf{l} & \mathsf{a} & \mathsf{y} & \mathsf{f} \\ \mathsf{i} & \mathsf{r} & \mathsf{b} & \mathsf{c} & \mathsf{d} \\ \mathsf{e} & \mathsf{g} & \mathsf{h} & \mathsf{k} & \mathsf{m} \\ \mathsf{n} & \mathsf{o} & \mathsf{q} & \mathsf{s} & \mathsf{t} \\ \mathsf{u} & \mathsf{v} & \mathsf{w} & \mathsf{x} & \mathsf{z} \end{matrix}$$

Primer 4.6: Kodirna matrika šifrirnika Playfair

Kodiranje poteka na naslednji način:

- iz vhodnega zaporedja I najprej odstranimo presledke;
- dobljeno zaporedje razdelimo v zaporedje parov črk I_p;
- če je par sestavljen iz dveh enakih črk, vrinemo črko x;
- črko x dodamo tudi črki, če ostane na koncu sama.

Šifriranje parov opravimo po naslednjih pravilih:

- če znaka nista v istem stolpcu ali v isti vrstici, zamenjamo vsako črko s črko, ki je v njeni vrstici in v stolpcu druge črke, nato pa zamenjamo črke v paru in ponovimo zamenjavo;

- če sta oba znaka v isti vrstici, zamenjamo vsako črko para z njeno desno črko, pri čemer pri zadnji črki v vrstici kot njeno desno sosedo vzamemo prvo črko v vrstici;

- če sta obe črki v istem stolpcu, zamenjamo črko para s črko, ki je v χ_{pf} pod črko. Črko iz zadnje vrstice matrike zamenjamo s črko iz prve vrstice v tem stolpcu.

Delovanje šifrirnika si bomo ogledali ob primeru sporočila v slovenskem jeziku, zato bomo postopek nekoliko prilagodili, in sicer:

- ker ima slovenska abeceda 25 črk, ni treba obravnavati črk i in j kot ene črke in

- ker slovenska abeceda nima črke x, bomo v primeru enakih znakov v paru črk vrinili črko f. Črko f bomo vrinili tudi, če na koncu ne bo para.

Naj bosta vhodno zaporedje I in ključ K takšna, kot kaže primer 4.7.

$$I = \langle \texttt{pod deblom hrasta je lok} \rangle$$

$$K = \langle \texttt{pri perunu} \rangle$$

Primer 4.7: Vhodno sporočilo in ključ za prikaz delovanja šifrirnika Playfair

Najprej tvorimo zaporedje parov I_p, ki jih kaže primer 4.8a, nato skonstruiramo šifrirno tabelo (glej primer 4.9). Tvorjeno šifro vidimo na primeru 4.8b.

(a) $I_p = \langle$`po df de bl om hr as ta je lo kf`$\rangle$
(b) $O = \langle$`ek fg hp am so fe če rf hr ms ed`$\rangle$

Primer 4.8: (a) Pari simbolov in (b) tvorjena šifra

$$\chi_{pf} = \begin{matrix} \mathsf{p} & \mathsf{r} & \mathsf{i} & \mathsf{e} & \mathsf{u} \\ \mathsf{n} & \mathsf{a} & \mathsf{b} & \mathsf{c} & \mathsf{č} \\ \mathsf{d} & \mathsf{f} & \mathsf{g} & \mathsf{h} & \mathsf{j} \\ \mathsf{k} & \mathsf{l} & \mathsf{m} & \mathsf{o} & \mathsf{s} \\ \mathsf{š} & \mathsf{t} & \mathsf{v} & \mathsf{z} & \mathsf{ž} \end{matrix}$$

Primer 4.9: Kodirna matrika χ_{pf} za ključ $K = \langle$priperunu$\rangle$

Dekodiranje poteka po obratni poti. Najprej iz ključa K sestavimo šifrirno matriko χ_{pf}, nato pa po parih iz O poiščemo z uporabo šifrirnih pravil dešifrirane pare.

4.2.2 Šifrirnik ADFGX

Šifrirnik ADFGX je iznašel poročnik Fritz Nebel, uporabljali pa so ga v nemški vojski med prvo svetovno vojno. Nemci so verjeli, da je šifrirnik varen, a v sredini leta 1918 ga je razvozlal francoski poročnik Georges Painvin. Kodirnik je dobil ime po črkah, ki so tvorile kodirano sporočilo. Te črke so izbrali tako, da jih je najlažje razpoznati, če jih kot Morsejeve znake sprejemamo *na uho* (A: $\cdot-$, D: $-\cdot\cdot$, F: $\cdot\cdot-\cdot$, G: $--\cdot$, X: $-\cdot\cdot-$). To izbiro lahko razumemo kot poskus kode ECC (angl. Error Correction Code). Šifrirnik ADFGX v svoji izvorni obliki očitno zmanjša abecedo. Abeceda vhodnega zaporedja I sestoji iz nemške abecede brez črk j, ß in črk s preglasom, izhodna abeceda Σ_O pa je $\Sigma_O = \{\mathsf{A\ D\ F\ G\ X}\}$. Posledično je $|I| < |O|$ oziroma velja: $|O| = 2|I|$. Kodirnik so pozneje razširili v kodirnik ADFGXV, ki je omogočal tudi šifriranje številk.

Šifrirnik ADFGX uporablja šifrirno matriko reda 5×5. Tokrat matriko ne sestavimo s pomočjo ključa, ampak je podana vnaprej. V ta namen so pripravili tajno kodirno knjigo s pripravljenimi šifrirnimi tabelami in pravila, kdaj eno izmed tabel uporabiti. Primer šifrirne matrike kaže razpredelnica 4.2.

Razpredelnica 4.2: Primer šifrirne tabele kodirnika ADFGX

	A	D	F	G	X
A	p	g	c	e	n
D	b	q	o	z	r
F	s	l	a	f	t
G	m	d	v	i	w
X	k	u	y	x	h

V nadaljevanju bomo tudi šifrirnik ADFGX priredili slovenski abecedi. Šifrirno tabelo s črkami slovenske abecede podaja razpredelnica 4.3, vhodno zaporedje pa kaže primer 4.10.

Tudi pri šifrirniku ADFGX ne šifriramo presledkov med besedami. Vsako črko vhodnega zaporedja najprej zamenjamo z oznakama vrstice in stolpca.

Razpredelnica 4.3: Primer šifrirne tabele kodirnika ADFGX s črkami slovenske abecede

	A	D	F	G	X
A	m	g	c	v	n
D	b	ž	o	k	r
F	s	l	a	f	t
G	p	d	e	i	š
X	z	u	č	j	h

$$I = \langle \texttt{knez kocelj} \rangle$$

Primer 4.10: Vhodno zaporedje za prikaz kodiranja ADFGX

Zaporedje šifriranih parov O_p, sestavljeno iz črk A, D, F, G, X, kaže primer 4.11.

$$O_p = \langle \texttt{DG AX GF XA DG DF AF GF FD XG} \rangle$$

Primer 4.11: Zaporedje šifriranih parov

Temu koraku sledi drugi korak, ki močno poveča moč šifre. Izberemo ključ, na primer $K = \langle metod \rangle$. Stolpce matrike označimo s črkami ključa, znake iz prvega koraka pa po vrsti vstavimo v matriko (glej primer 4.12).[1]

m	e	t	o	d
D	G	A	X	G
F	X	A	D	G
D	F	A	F	G
F	F	D	X	G

Primer 4.12: Drugi korak šifrirnika ADFGX uporabi ključ *metod*.

Stolpce matrike nato razvrstimo po abecednem vrstnem redu, da dobimo stanje, ki ga kaže primer 4.13.

Nazadnje znake iz stolpcev po vrsti izpišemo, da dobimo končno šifro O, ki jo kaže primer 4.14. Znake šifre običajno v slogu telegrafskih skupin združimo v bloke po 5 znakov. Dešifriranje poteka po obratnem vrstnem redu, seveda pa moramo poznati ključ in šifrirno tabelo. Iz dolžine ključa in dolžine sporočila določimo število in dolžino stolpcev. Znake ključa uredimo po abecednem redu in stolpce napolnimo z znaki šifre. Stolpce nato

[1] V splošnem se lahko število znakov v stolpcih razlikuje za 1.

d	e	m	o	t
G	G	D	X	A
G	X	F	D	A
G	F	D	F	A
G	F	F	X	D

Primer 4.13: Urejanje znakov ključa.

$O = \langle$GGGGG XFFDF DFXDF XAAAD$\rangle$

Primer 4.14: Končna šifra sporočila ⟨knez kocelj⟩.

prerazporedimo glede na ključ. Pare znakov zatem uporabimo za to, da iz matrike odberemo znake sporočila.

Naloge

1. S pomikalnim šifrirnikom zakodirajte sporočilo $I = \langle$le vkup le vkup uboga gmajna$\rangle$, pri čemer naj bo $K = 14$. Šifrirano sporočilo nato dekodirajte.

2. S pomikalnim šifrirnikom zakodirajte sporočilo $I = \langle$submarine is on the way$\rangle$, kjer je $K = 12$. Tudi tokrat dobljeno šifro dekodirajte.

3. Napišite program, ki bo na vhodu sprejel abecedo, sporočilo in ključ ter opravil šifriranje in dešifriranje s pomikalnim šifrirnikom.

4. Napišite program, ki bo na vhodu sprejel abecedo in šifro ter s poskušanjem poiskal ključ K pri pomikalnem šifrirniku.

5. Z Vigenérjevim šifrirnikom zakodirajte sporočilo $I = \langle$za mavrico se potrebujeta tako dež kot sonce$\rangle$, pri čemer uporabite ključ $K = \langle$grom$\rangle$. Šifrirano sporočilo nato dešifrirajte.

6. Sporočilo $I = \langle$railway is broken at the longbridge$\rangle$ zašifrirajte z Vigenérjevim šifrirnikom s ključem $K = \langle$explosion$\rangle$. Tudi tokrat sporočilo dešifrirajte.

7. Napišite program, ki bo na vhodu sprejel abecedo, sporočilo in ključ ter opravil šifriranje in dešifriranje z Vigenérjevim šifrirnikom.

8. Napišite program, ki bo na vhodu sprejel abecedo in šifro, nato pa s poskušanjem poiskal ključ za Vigenérjev šifrirnik. Do kakšne dolžine ključa bo deloval algoritem v doglednem času, če je abeceda ASCII?

9. Zamislite si program, ki bo idejo Vigenérjevega šifrirnika uporabil za prijavo v vaše aplikacije, pri čemer naj uporabnik poda (in pomni) ključ za Vigenérjev šifrirnik.

10. Naj bo ključ $K = \langle$veliki traven$\rangle$. Sporočilo $I = \langle$kosci že navsezgodaj brusijo svoje kose$\rangle$ zakodirajte s kodirnikoma Playfair in ADFGX. Pravilnost šifre preverite z dešifriranjem.

11. Naj bo ključ $K = \langle$walker$\rangle$. Sporočilo $I = \langle$each journey starts with the first step$\rangle$ zakodirajte s kodirnikom Playfair. Dobljeno šifro dešifrirajte.

12. Naj bo ključ $K = \langle$uboot$\rangle$. Sporočilo $I = \langle$der konvoi fahrt am mittwoch ab$\rangle$ zakodirajte s kodirnikom ADFGX. Pridobljeno šifro dešifrirajte.

13. Napišite program, ki bo s podano abecedo in ključem zašifriral sporočilo po metodah Playfair in ADFGX.

14. Razmislite, kako bi dekodirali sporočilo, zakodirano s šifrirnikom Playfair, če ne poznate ključa? Napišite program, ki bo iskanje ključa tudi opravil.

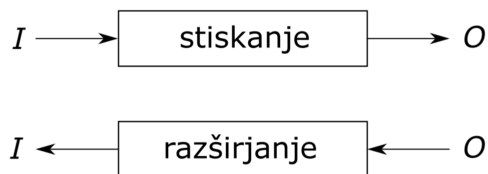
15. Podobno kot v prejšnji nalogi razmislite, kako bi dekodirali šifro ADFGX.

Poglavje 5

Stiskanje podatkov

Stiskanje podatkov (angl. data compression) je eno izmed najstarejših področij računalništva, ki se je pričelo z deli Shannona [27], Fanoja [28] in Huffmana [29]. Pri številnih aplikacijah ima ključno vlogo, saj bi zaradi količine tako imenovanih **surovih podatkov** bila njihova uporaba zelo omejena ali celo nemogoča. Poglejmo ilustrativen primer. Fotografija, ki je predstavljena kot rastrska slika, naj ima velikost 1.024×1.024 pikslov, vsak piksel pa je opisan s tremi zlogi (24 biti). Surovi podatki za opis pikslov bi tako zahtevali 25.165.824 bitov, kar je 3.145.728 zlogov oziroma 3 MiB. Iz izkušenj pa vemo, da so takšne slike zapisane z bistveno manj zlogi. Brez znanja o predstavitvi informacij v jedrnati obliki bi danes svet bil bistveno drugačen. Digitalna televizija še ne bi delovala, vsebine na svetovnem spletu pa bi bile pustejše.

Stiskanje podatkov lahko neformalno obravnavamo kot proces pretvorbe vhodnega zaporedja I v izhodno zaporedje O tako, da je število bitov $|O|$ manjše, kot smo jih potrebovali za zapis zaporedja I, torej $|I| > |O|$. Proces mora biti povraten; iz stisnjenih podatkov mora biti možno rekonstruirati začetno zaporedje (slika 5.1); postopek bomo imenovali **razširjanje** podatkov.



Slika 5.1: Proces stiskanja in razširjanja je povraten

Uspešnost stiskanja podatkov želimo pogosto meriti. V literaturi se pojavljajo različne definicije [30]:

- **razmerje stiskanja** (angl. compression ratio)

$$CR = \frac{|O|}{|I|}, \tag{5.1}$$

- **faktor stiskanja** (angl. compression factor)

$$CF = \frac{|I|}{|O|}, \tag{5.2}$$

- **prihranek stiskanja** (angl. saving percentage)

$$SP = 100\frac{|I| - |O|}{|I|} = 100(1 - CR). \tag{5.3}$$

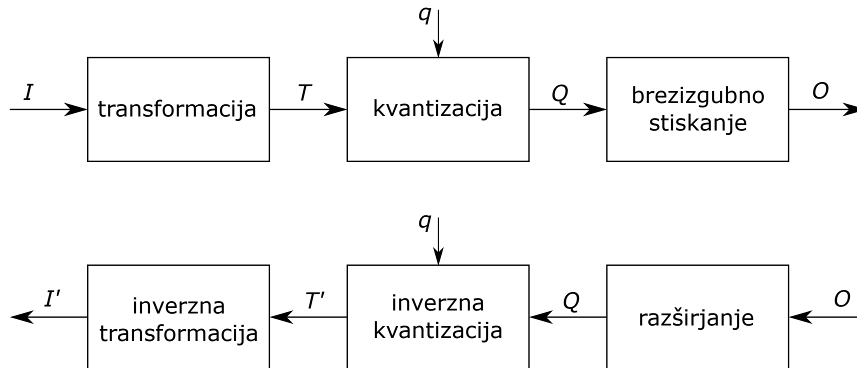
Za stiskanje podatkov je bilo razvitih veliko algoritmov. Odlične opise najdemo v številnih knjigah [30, 31, 32, 33]. Pristope lahko delimo na več načinov, a najbolj splošni sta:

- splošnonamenske in domenskoodvisne metode ter
- izgubne in brezizgubne metode.

Splošnonamenske metode poskusijo stisniti podatke, ne da bi vnaprej poznale zakonitosti, ki se morebiti skrivajo v podatkih. Prav obratno pa domenskoodvisne metode izkoriščajo znanje o podatkih za doseganje večjega prihranka. Na primer poznavanje zakonitosti o strukturi datoteke HTML lahko uporabimo kot zasnovo učinkovitejšega algoritma stiskanja [34].

Druga pogosta delitev je delitev na metode, ki stiskajo podatke brez izgub, in tiste, ki vnašajo izgube. Vseh tipov podatkov seveda ne smemo stiskati z izgubami. Izgube pri stiskanju besedila so popolnoma nesprejemljive. Tudi medicinskih slik ne bomo stiskali z izgubami. Izgube si lahko privoščimo le tam, kjer naša čutila ne zmorejo slediti kakovosti izvornih podatkov (na primer nekatere vrste rastrskih slik, digitalna glasba in digitalni video). Izgubne metode vključujejo vsaj tri korake, kot kaže slika 5.2. Vhodno zaporedje podatkov I najprej transformiramo v prostor, v katerem lažje ugotovimo, kateri koeficienti so bolj in kateri manj pomembni. Manj pomembne nato kvantiziramo (postavimo na vrednost 0). Velikost kvantizacije določi uporabnik s koeficientom q in z njim nadzira količino izgub.

Kvantizirano zaporedje nato brezizgubno stisnemo v izhodno zaporedje O. Razširjanje podatkov opravimo v obratnem vrstnem redu. Zaradi kvantizacije podatkov ne povrnemo popolnoma, zato se razširjeni podatki nekoliko razlikujejo od izvornih, to je $I' \neq I$, a razlika je takšna, da je človek s svojimi čutili praviloma ne zazna.



Slika 5.2: Tipični cevovod izgubnega stiskanja

V nadaljevanju si bomo ogledali različne metode brezizgubnega stiskanja. Preden se lotimo pravih metod, si oglejmo intuitivne pristope, ki bi jih najverjetneje izdelali brez predhodnega znanja o stiskanju podatkov [30].

5.1 Intuitivne metode stiskanja

Nekaj intuitivnih metod stiskanja povzemamo po [30]:

- Naj bo vhodno zaporedje I iz primera 5.1a. Če vsak znak v zaporedju zakodiramo z razširjenim naborom ASCII, potrebujemo 33 zlogov oziroma 264 bitov. V stavku je 5 presledkov. Če bi jih izpustili, bi dobili izhodno zaporedje O1, ki bi mu sledilo zaporedje bitov $O2$ (glej primer 5.1b), kjer smo vsak znak, ki ni presledek, označili z bitom 0, presledek pa z bitom 1. Stisnjeno zaporedje O dobimo z lepljenjem zaporedij $O1$ in $O2$. Tako zakodirano sporočilo bi imelo $28 \cdot 8 + 33 = 257$ bitov, kar je 7 bitov manj kot v surovi obliki. Dobljeno razmerje stiskanja je 0,9735 in je zelo odvisno od dolžine besed.

- Namesto 8-bitnih kod ASCII bi lahko sporočilo I zakodirali s 7-bitnim naborom ASCII. Za takšen zapis bi potrebovali $33 \cdot 7 = 231$ bitov. Razmerje stiskanja je tokrat neodvisno od sporočila in je $\frac{7}{8} = 0{,}875$.

a) $I = \langle$`Vsaka ideja ni vedno dobra ideja.`$\rangle$

b) $O1 = \langle$`Vsakaidejanivednodobraideja.`$\rangle$
$O2 = \langle$0000010000010010000010000010000000$\rangle$

Primer 5.1: (a) Sporočilo I in (b) stiskanje s kodiranjem presledkov

- Urejevalnik MacWrite je 15 najpogostejših znakov ␣etnroaisdlhcfp angleške abecede in posebno kontrolno kodo zakodiral s štirimi biti, za druge znake pa je uporabil kontrolno kodo, ki ji je sledila koda ASCII posameznega znaka (skupaj torej 12 bitov). Vsak odstavek kodiramo ločeno. Če bi rezultat kodiranja vodil v razširjanje, odstavek zapišemo v ASCII. Na začetku vsakega odstavka zato dodamo bit, ki pove, ali uporabljamo *stiskanje* ali ne.

- Podatke, ki jih zapišemo kot leksikografsko urejeno zaporedje (na primer slovar), lahko stisnemo s principom **čelnega stiskanja** (angl. front compression). V tem primeru velja, da si zaporedne besede delijo začetne znake. Stiskanje dosežemo, če prvih n ujemajočih se znakov zamenjamo z dolžino ujemanja. Poglejmo si primer 5.2.

baba	baba
babica	3ica
babuška	3uška
bala	2la
balkan	3kan
balkanski	6ski
beg	1eg
begunje	3unje
begunec	5ec
bencin	2ncin
ciciban	0ciciban
cicido	4do

Primer 5.2: Čelno stiskanje

5.1.1 Stiskanje zaporedja enakih znakov

Stiskanje zaporedja enakih simbolov (angl. Run-Length Encoding, RLE) je sicer intuitivna tehnika, ki pa jo pogosto uporabljamo, kar kaže, da intuicija ni nujno vedno slaba. V enem od korakov ga uporablja tudi standard za stiskanje slik JPEG [35]. RLE temelji na predpostavki, da se v zaporedju, ki ga stiskamo, isti simboli pogosto ponavljajo zaporedoma. Zato v izhodno zaporedje zapišemo simbol, ki mu sledi podatek o številu njegovih ponovitev, kot kaže primer 5.3.

(a) $I = \langle \texttt{aabbbbccaaaaaaabbcccccc} \rangle$
(b) $\Sigma = \{\texttt{a,b,c}\}$

(c) $O = \langle \texttt{a1b3c1a6b1c5} \rangle$

Primer 5.3: (a) Vhodni niz, (b) abeceda in (c) stisnjen niz z RLE

Tehnika žal lahko vodi v razširjanje podatkov, če je ponavljanj malo (glej primer 5.4).

$$I = \langle \texttt{abccbaaaabc} \rangle$$
$$O = \langle \texttt{a0b0c1b0a3b0c0} \rangle$$

Primer 5.4: Če ponavljanj zaporednih znakov ni veliko, RLE odpove

Ena od rešitev je, da vstopimo v način RLE le takrat, ko se to morebiti obrestuje, sicer pa zapisujemo simbole v nestisnjeni obliki (na primer s kodami ASCII). Za to potrebujemo tako imenovani ubežni simbol (angl. *escape symbol*), s katerim preklapljamo med načini kodiranja.

Naj bo zaporedje $X = \langle \sigma_{\mathrm{i}} \rangle$, $\sigma_i \in \Sigma$, kjer je Σ abeceda in σ_i njeni simboli. **Ubežni simbol** σ_{Esc} je lahko vsak simbol, za katerega velja: $\sigma_{Esc} \notin \Sigma$. Stanje, ko je $\sigma_{Esc} =!$, vidimo kaže primer 5.5.

$$I = \langle \texttt{abbbbcaaaaaaabbcccccc} \rangle$$
$$O = \langle \texttt{a!b3c!a6!b1!c5} \rangle$$

Primer 5.5: Vstop v način RLE nadziramo s simbolom $\sigma_{ESC} =!$

Tudi ta rešitev ima svoje slabosti, saj potrebujemo dodatni simbol. Ugodnejša in pogosteje uporabljana možnost za vstop v način RLE je zaporedje t enakih znakov. V tem primeru kodiranje RLE nadziramo s tremi parametri: t, σ_i in l_{σ_i}, kjer so:

- t: globalni parameter, ki določa število zaporednih znakov za vstop v način RLE,
- σ_i: znak, ki se ponavlja in
- l_{σ_i}: število ponavljanj znaka σ_i.

Ker je t globalni parameter, ga shranimo samo enkrat, žeton, ki nadomešča zaporedje ponavljajočih se znakov, pa predstavljata parametra (σ_i, l_{σ_i}). Primer stiskanja s to različico RLE, ko je $t = 2$, kaže primer 5.6.

$$I = \langle \texttt{abbbbcaaaaaaabbcccccc} \rangle$$
$$O = \langle \texttt{ab2ca5b0c4} \rangle$$

Primer 5.6: Vstop v način RLE nadziramo z globalnim parametrom t

Pogosto se v zaporedju ponavlja samo en simbol, najpogosteje 0. V tem primeru je žeton krajši in sestoji samo iz števila ponavljanj n. Takšnemu kodiranju RLE pravimo kodiranje ZRT (angl. Zero-Run Transform) [36] (glej primer 5.7).

$$I = \langle \texttt{0b00b0000b00000a} \rangle$$
$$O = \langle \texttt{0b000b002b003a} \rangle$$

Primer 5.7: Kodiranje ZRT, ko je $t = 2$

Na žalost so zaporedja enakih znakov v besedilih zelo redka, zato neposredna uporaba te tehnike pri stiskanju besedil odpove. V kombinaciji z Burrows-Wheelerjevo transformacijo in transformacijo premik naprej pa je RLE oz. celo ZRT zelo uporaben (Burrows-Wheelerjevo transformacijo in transformacijo premik naprej bomo spoznali v poglavju 6). Neposredna uporaba tehnike RLE je pogostejša pri rastrskih slikah, kjer lahko pričakujemo daljša zaporedja pikslov enakih barv. To tehniko neposredno uporabljata formata PCX in BMP, slednji v načinu z barvno preslikovalno razpredelnico (angl. lookup table, LUT).

5.2 Entropija informacije

Za vsako zaporedje nas zanima, kako močno bi ga lahko stisnili. Ali je vsako zaporedje sploh stisljivo? Odgovore na to nam da **entropija informacije**. V nadaljevanju bomo njen pomen razložili neformalno.

Naj bo X slučajna spremenljivka z neko zalogo možnih vrednosti x_i, $0 \leq i < n$. Da bi izvedeli, kakšno vrednost je spremenljivka X zavzela, moramo opraviti poizkus. S ponavljanjem poizkusa lahko za vsako vrednost x_i, ki jo more zavzeti X, določimo verjetnost p_i njene ponovitve. Prav poseben primer dobimo, ko lahko naključna spremenljivka zavzame le dve vrednosti. Takšen poizkus je tudi met kovanca. Če je kovanec pošten, je verjetnost izida poizkusa enaka za obe strani kovanca, to je $p_{glava} = p_{pismo} = 0{,}5$. V tem primeru pravimo, da nam vsak izid poizkusa da 1 bit informacije. Pri napovedovanju izida poizkusa pri metu nepoštenega kovanca bomo, če vemo, na katero stran kovanec raje pade, seveda uspešnejši. Posledično se količina informacije po opravljenem poizkusu zmanjša. Ko lahko naključna spremenljivka zavzame le dve vrednosti, govorimo o **informacijski entropiji** ali **Shannonovi entropiji**, ki jo izračunamo z enačbo 5.4.

$$H(X) = -\sum_{0}^{n-1} p_{x_i} \log_2 p_{x_i} \tag{5.4}$$

Sporočilo $I = \langle \sigma_i \rangle$ (primer 5.8a) je sestavljeno iz simbolov abecede Σ, $\sigma_i \in \Sigma$ (primer 5.8b). Oglejmo si, kaj nam sporoča enačba 5.4. V konkretnem primeru abeceda Σ ustreza zalogi vrednosti neodvisne spremenljivke X. Vidimo, da se simboli v I pojavljajo različno pogosto; $\sigma_0 = \texttt{a}$ se pojavi

(a) $I = \langle \texttt{abcaaaabba} \rangle$
(b) $\Sigma = \{\ \texttt{a}\quad \texttt{b}\quad \texttt{c} \}$
(c) $P = [0{,}6\ \ 0{,}3\ \ 0{,}1]$

Primer 5.8: (a) Vhodno zaporedje, (b) abeceda in (c) verjetnosti pojavljanja simbolov

šestkrat, $\sigma_1 = \texttt{b}$ trikrat in $\sigma_2 = \texttt{c}$ samo enkrat, iz česar določimo vektor verjetnosti pojavitve simbolov P (glej primer 5.8c). Shannonova entropija za zaporedje I je tako

$$\begin{aligned} H(I) &= -(0{,}6\ \log_2(0{,}6) + 0{,}3\ \log_2(0{,}3) + 0{,}1\ \log_2(0{,}1)) = \\ &= -(-0{,}44218 - 0{,}52109 - 0{,}33219) = 1{,}29546. \end{aligned}$$

Primer 5.9: Izračun informacijske entropije

Rezultat izračuna (primer 5.9) pomeni, da bi v idealnem primeru za zapis tega sporočila potrebovali 1,29546 bita na simbol; ker je $|I| = 10$, bi celotno sporočilo zapisali z $12{,}9546 \approx 13$ biti. Če bi sporočilo I zapisali s

kodo ASCII, bi za zapis porabili 80 bitov (8 bitov na simbol), kar je zelo potratno. Da bi dosegli jedrnatejši zapis sporočila I, bomo morali simbolom

Razpredelnica 5.1: Dodelitev bitov simbolom – prva možnost

σ_i	koda
a	0
b	10
c	11

$\sigma_i \in \Sigma$ dodelili drugačne binarne kode. Eno od možnosti kaže razpredelnica 5.1, kjer smo simbolu a z največjo verjetnostjo dodelili najkrajšo kodo. Ko imajo simboli prirejene kode različnih dolžin, govorimo o **kodi s spremenljivo dolžino** (angl. variable length code, VLC). Če bi sporočilo I iz primera 5.8 zapisali s kodami iz razpredelnice 5.1, bi dobili zaporedje O iz primera 5.10. Ugotovimo, da je $|O| = 14$ bitov, kar je veliko bolje, kot če bi

$$O = \langle \mathsf{01011000010100} \rangle$$

Primer 5.10: Jedrnatejši zapis sporočila I s kodami iz razpredelnice 5.1

sporočilo zapisali s kodami ASCII, a nekoliko slabše, kot pravi izračun iz primera 5.9; za zapis sporočila I potrebujemo namreč 1,4 bita na simbol. Eden od razlogov, da nismo dosegli vrednosti Shannonove entropije je zagotovo ta, da bitov ne znamo deliti, drugi pa je morda povezan z načinom dodelitve kod našim simbolom. Na prvi pogled ugodnejšo dodelitev kod simbolom vidimo v razpredelnici 5.2, ki nam tvori izhodno zaporedje O dolžine samo 11 bitov (glej primer 5.11) oziroma samo 1,1 bita na simbol, kar je celo bolje, kot smo izračunali s Shannonovo enačbo. Žal pa hitro ugotovimo, da takšen izbor kod ne omogoča enoličnega dekodiranja. Pravimo, da izbrane kode niso predponske/prefiksne (angl. prefix code).

Koda je predponska, če nobena samostojna koda ni enaka predponi neke druge kode. Samo predponske kode lahko enolično tudi dekodiramo [30].

V našem primeru je bit 1, ki predstavlja kodo simbola b, tudi predpona kodi 10, ki predstavlja znak c. Ko imamo zaporedje bitov 10, ga lahko dekodiramo kot zaporedje simbolov $\langle \mathsf{ab} \rangle$ ali simbol c – dekodiranje torej ni enolično.

Razpredelnica 5.2: Dodelitev bitov simbolom – druga možnost

σ_i	koda
a	0
b	1
c	10

$$O = \langle \texttt{01100000110} \rangle$$

Primer 5.11: Zapis sporočila I s kodami iz razpredelnice 5.2

Razpredelnica 5.3: Še dve možnosti dodelitve bitov simbolom iz sporočila I

σ_i	koda
a	10
b	11
c	0

(a)

σ_i	koda
a	00
b	01
c	10

(b)

S poskušanjem lahko določimo našim simbolom iz Σ še drugačne bitne kode; dve možnosti vidimo v razpredelnici 5.3, dobljeni izhodni zaporedji pa kaže primer 5.12. Obe kodi sta predponski, saj lahko bitni zaporedji enolično dekodiramo. Zaporedje iz primera 5.12a ima 19 bitov, zaporedje iz primera 5.12b pa 20 bitov, kar je občutno slabše, kot smo dosegli s kodami iz razpredelnice 5.1.

(a) $O = \langle \texttt{1011010101010111110} \rangle$
(b) $O = \langle \texttt{00011000000000010100} \rangle$

Primer 5.12: Zapis sporočila I s kodami iz razpredelnice 5.3

Ugotovimo torej, da se lahko s smiselno izbiro predponskih kod približamo teoretični spodnji meji števila bitov na simbol, ki jo določa enačba 5.4. Pri majhnem številu simbolov lahko simbolom dodelimo najoptimalnejše kode z logičnim razmislekom, kot smo to lahko storili v našem primeru, ko je bila $|\Sigma| = 3$. Pri večjem številu simbolov pa bomo hitro v zagati, ali nas izbira kod zares dovolj približa Shannonovi entropiji. Preden si bomo ogledali postopke, kako to doseči, se pozabavajmo še s kakšnim vidikom Shannonove entropije.

$$\text{(a)} \quad \begin{aligned} I &= \langle \texttt{abcabcabc} \rangle \\ \Sigma &= \{\texttt{a b c}\} \\ P &= [\tfrac{1}{3}\ \tfrac{1}{3}\ \tfrac{1}{3}] \end{aligned}$$

$$\text{(b)} \quad \begin{aligned} I &= \langle \texttt{abbbbbbcc} \rangle \\ \Sigma &= \{\texttt{a b c}\} \\ P &= [\tfrac{1}{9}\ \tfrac{6}{9}\ \tfrac{2}{9}] \end{aligned}$$

Primer 5.13: Zaporedje z (a) enakimi in (b) zelo različnimi verjetnostmi pojavitve simbolov

Kakšna bi bila Shannonova entropija pri sporočilih iz primerov 5.13a in 5.13b? Število simbolov je v obeh primerih enako, prav tako abeceda, razlika je le v verjetnosti pojavitve simbolov. V primeru sporočila 5.13a dobimo informacijsko entropijo $H(I) = 1{,}58496$, v primeru 5.13b pa $H(I) = 1{,}22439$, kar je občutno manj. Iz tega primera upravičeno sklepamo, da dosežemo boljše stiskanje takrat, ko se verjetnosti pojavljanja simbolov čim bolj razlikujejo. V primeru 5.13a je popolnoma vseeno, kateremu simbolu bi dodelili samo en bit, v primeru 5.13b pa, kot smo spoznali, to zagotovo ne velja.

Ob koncu še razmislimo, ali je vsako zaporedje tudi stisljivo. Primer 5.14 kaže zaporedje I, sestavljeno iz štirih simbolov. Intuitivno simbolom priredimo dvobitne kode, ki so v razpredelnici 5.4. Če izračunamo informacijsko entropijo zaporedja I, ugotovimo, da je $H(I) = 2$, torej so kode že optimalne in sprememba kodiranja ne bi pripomogla k zmanjšanju števila bitov.

$$\begin{aligned} I &= \langle \texttt{abcdbacdcbda} \rangle \\ \Sigma &= \{\texttt{a b c d}\} \\ P &= [0{,}25\ \ 0{,}25\ \ 0{,}25\ \ 0{,}25] \end{aligned}$$

Primer 5.14: Zaporedje s štirimi simboli, katerih verjetnost je enaka

5.3 Statistično stiskanje podatkov

Statistično stiskanje podatkov temelji na pogostosti pojavljanja simbolov. Ločimo jih v dve veliki družini, in sicer algoritmi, ki dodelijo simbolom kode VLC, in algoritmi, ki simbolom dodelijo interval na številski premici. V prvo skupino spadata Shannon-Fanojevo in Huffmanovo kodiranje ter njune

Razpredelnica 5.4: Dodelitev bitov štirim, enako verjetnim, simbolom

σ_i	koda
a	00
b	01
c	10
d	11

različice, v drugo pa aritmetično kodiranje z množico izpeljank.

5.3.1 Shannon-Fanojev algoritem

Čeprav je algoritem izumil Fano davnega leta 1949 [28], ga v literaturi najpogoste srečamo pod imenom Shannon-Fanojev algoritem [30, 32]. Kodiranje ima naslednje lastnosti:

- različni simboli σ_i vhodnega zaporedja $I = \langle \sigma_i \rangle$ dobijo kode VLC,
- kode simbolov, ki se redkeje pojavijo, imajo več bitov, kode pogostejših simbolov pa so krajše,
- tvorjene kode so predponske, zato je dekodiranje enolično.

Shannon-Fanojev algoritem med postopkom določitve kod VLC zgradi drevo $\mathcal{T}$, ki ga uporabimo tudi za dekodiranje, zato ga imenujemo **kodirno-dekodirno drevo**. Določitev Shannon-Fanojevih kod dobimo z naslednjim postopkom:

1. Ugotovimo, kateri simboli σ_i so v vhodnem sporočilu $I = \langle \sigma_i \rangle$ dolžine $n = |I|$. Ti simboli sestavljajo abecedo Σ.
2. Za vsak $\sigma_i \in \Sigma$ določimo njegovo pogostost pojavljanja v sporočilu. Pravimo, da določimo njegovo frekvenco f_i.
3. Simbole $\sigma_i \in \Sigma$ in njim pripadajoče frekvence f_i vstavimo v zaporedje, ki ga uredimo glede f_i, $f_i \geq f_{i+1}$, $0 \leq i < n$. To zaporedje ustreza korenu $\mathcal{T}$.
4. Zaporedje razdelimo na dva dela tako, da je seštevek frekvenc v obeh podzaporedjih čim bolj enak. Podzaporedje z večjimi frekvencami predstavlja levega potomca $\mathcal{T}$, drugi pa desnega.

5. Veji, ki kaže na levega potomca, priredimo bit 0, desni veji pa bit 1.

6. Deljenje ponavljamo rekurzivno, dokler zaporedje ni sestavljeno iz samo enega simbola, ki postane list $\mathcal{T}$.

7. Kodo VLC za posamezni σ_i dobimo z lepljenjem bitov, ki so na vejah $\mathcal{T}$ ob sprehodu od korena do lista drevesa.

Shannon-Fanojev algoritem razložimo še s primerom 5.15.

$$I = \langle \texttt{ddeeaebebcecdbdeaaaeebbeaedcdcebaeebecec} \rangle$$
$$\Sigma = \{\texttt{a,b,c,d,e}\}$$

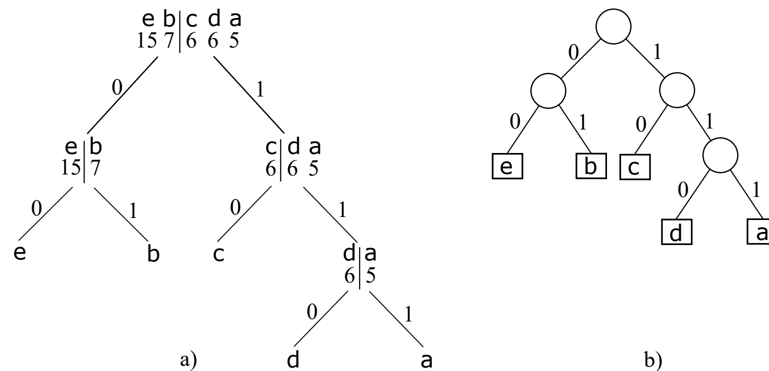
Primer 5.15: Vhodno zaporedje za določitev kod VLC

Iz I hitro ugotovimo frekvence posameznih simbolov, ki jih s pripadajočimi simboli uredimo glede na f_i (glej primer 5.16).

σ_i:	e	b	c	d	a
f_i:	15	7	6	6	5

Primer 5.16: Simboli sporočila σ_i, urejeni glede na pripadajoče frekvence f_i

Slika 5.3a ponazarja postopek deljenja zaporedja I, kodirno-dekodirno drevo $\mathcal{T}$ pa vidimo na sliki 5.3b. Shannon-Fanojeve kode za vse $\sigma_i \in \Sigma$ našega primera so zbrane v razpredelnici 5.5.



Slika 5.3: (a) Deljenje zaporedja in (b) Shannon-Fanojevo kodirno-dekodirno drevo

Simbole $\sigma_i \in I$ po vrsti zamenjamo z ustreznim naborov bitov. Dobimo izhodno bitno zaporedje O (glej primer 5.17).

Razpredelnica 5.5: Simboli in Shannon-Fanojeve kode

σ_i	e	b	c	d	a
koda	00	01	10	110	111

$$O = \langle 11011000000111000100011000101100111000111111100$$
$$000101001110011010110100001111000000100100010\rangle$$

Primer 5.17: Zaporedje bitov Shannon-Fanojevih kod

Bite bomo od koncu shranili v datoteko, ki pa jo moramo sestaviti tako, da bo stisnjeno sporočilo možno tudi razširiti. Zato datoteko opremimo z glavo datoteke, ki ji sledi zaporedje bitov. V glavo datoteke zapišemo: $|\Sigma|$, znake abecede σ_i in njihove frekvence f_i. Za naš primer vidimo organizacijo datoteke z glavo in bitnim zaporedjem na sliki 5.4.

glava	niz bitov		
5 a b c d e 5 7 6 6 15	11011000	...	100010

Slika 5.4: Organizacija datoteke s stisnjenimi podatki

Razširjanje je preprosto. Najprej preberemo podatke iz glave datoteke ter po opisanem postopku sestavimo $\mathcal{T}$, nato začnemo dekodirati. Postavimo se v koren $\mathcal{T}$ ter preberemo bit iz datoteke. Če je bit 0, se premaknemo do levega potomca, sicer pa do desnega. Če smo prišli v notranje vozlišče $\mathcal{T}$, preberemo naslednji bit, sicer pa izpišemo σ_i, ki je v listu drevesa. Nato se ponovno postavimo v koren drevesa in postopek ponavljamo, dokler ne dosežemo konca datoteke.

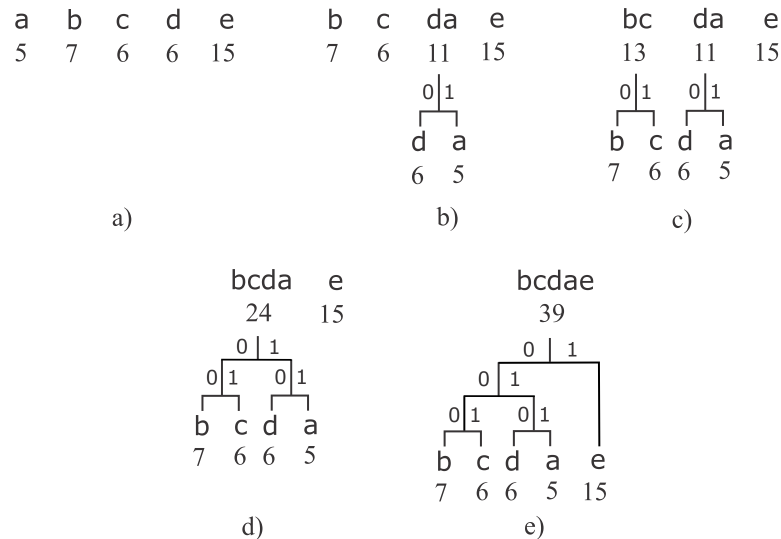
5.3.2 Huffmanov algoritem

D. A. Huffman je leta 1952 objavil najbolj znan postopek določanja kod VLC za stiskanje podatkov. Huffmanov algoritem je zelo podoben Shannon-Fanojevemu: simboli z večjo verjetnostjo prejmejo krajše kode, kode so predponske, dekodiranje opravimo s kodirno-dekodirnim drevesom $\mathcal{T}$. Sam postopek gradnje $\mathcal{T}$ pa se razlikuje od Shannon-Fanojevega. Medtem ko pri Shannon-Fanojevem algoritmu gradimo $\mathcal{T}$ od zgoraj navzdol, je pri Huffmanovi metodi pot nasprotna, saj gradnja $\mathcal{T}$ poteka od simbolov σ_i proti korenu. Postopek je naslednji:

1. Ugotovimo, kateri simboli σ_i so v vhodnem sporočilu $I = \langle\sigma_i\rangle$ in določimo abecedo sporočila Σ.

2. Za vsak $\sigma_i \in \Sigma$ določimo frekvenco pojavljanja f_i.
3. Simbole σ_i z njihovimi frekvencami f_i vstavimo v seznam; simbolom, ki so v seznamu, pravimo **prosta vozlišča**.
4. Poiščemo dve prosti vozlišči z najmanjšima frekvencama f_i in f_j (ta trenutek spreglejmo možnost, da je takšnih vozlišč lahko več).
5. Za ti dve vozlišči ustvarimo starša. Tudi staršu določimo frekvenco, in sicer kot vsoto frekvenc njegovih potomcev, $f_s = f_i + f_j$.
6. Vejama, ki vodita do potomcev, priredimo bit `1` oziroma `0`.
7. Starša uvrstimo v seznam prostih vozlišč, oba potomca pa iz tega seznama odstranimo.
8. Korake 4, 5, 6 in 7 ponavljamo tako dolgo, dokler ne ostane samo eno prosto vozlišče, ki je koren Huffmanovega drevesa $\mathcal{T}$.
9. S sprehodom od korena do listov $\mathcal{T}$ določimo binarne kode VLC za vse simbole $\sigma_i \in \Sigma$. Tem kodam pravimo tudi **Huffmanove kode**.

Tudi tokrat ponazorimo algoritem s primerom. Da bomo lažje primerjali Huffmanove kode s Shannon-Fanojevimi, uporabimo zaporedje iz primera 5.15.



Slika 5.5: Postopek gradnje Huffmanovega kodirno-dekodirnega drevesa

Najprej ustvarimo seznam prostih vozlišč (slika 5.5a). Poiščemo vozlišči z najmanjšima frekvencama (v terminologiji Huffmanovega algoritma govorimo tudi o težah). V našem primeru obstajata dva takšna para (d in a ter c in a). Izberemo enega od njiju (predpostavimo, da smo se odločili za d in a). Tvorimo njunega starša, ki dobi frekvenco $f_s = 11$. Robova do vozlišč d in a označimo z bitoma 0 in 1 (Huffman sicer ni določil, kako ju izberemo). Odločimo se, da bomo potomcu z manjšo frekvenco dali bit 1, potomcu z večjo pa 0. Vozlišči d in a s seznama prostih vozlišč odstranimo, vstavimo pa njunega starša. Dobimo situacijo, ki jo vidimo na sliki 5.5b. V naslednjem koraku vzamemo vozlišči b in c, ki imata najmanjši vrednosti frekvenc. Njun starš dobi frekvenco $f_s = 13$ (slika 5.8c). Zatem združimo prosti vozlišči bc in da, saj imata najmanjši frekvenci. Skupna frekvenca je 24 (slika 5.5d). V zadnjem koraku sta ostali samo še dve prosti vozlišči, ki ju združimo. Njun starš ima frekvenco 39. Po odstranitvi potomcev iz seznama prostih vozlišč ostane v seznamu samo eno vozlišče, zato postopek zaključimo. Huffmanovo kodirno-dekodirno drevo $\mathcal{T}$ vidimo na sliki 5.5e, v razpredelnici 5.6 pa zbrane Huffmanove kode.

Razpredelnica 5.6: Huffmanove kode

σ_i	a	b	c	d	e
koda	011	000	001	010	1

Če primerjamo kode VLC, ki smo jih dobili s Huffmanovim in Shannon-Fanojevim postopkom (razpredelnici 5.5 in 5.6), ugotovimo:

- Huffmanov algoritem priredi simbolu e en bit, Shannon-Fannojev pa dva;
- simbola b in c sta pri Shannon-Fanojevemu kodiranju predstavljena z dvema bitoma, pri Huffmanovem pa s tremi;
- simbola a in d sta pri obeh kodiranjih zapisana s tremi biti.

Katero kodiranje bi bolje stisnilo sporočilo iz primera 5.15? Enostaven izračun (glej primer 5.18) nam da odgovor. V tem primeru je Huffmanovo kodiranje učinkovitejše od Shannon-Fanojevega za dva bita. To sicer ni veliko (več kot 2 %), a je sporočilo tudi zelo kratko. Izkaže se, da Huffmanova metoda v večini primerov da krajšo izhodno bitno zaporedje, zato Shannon-Fanojevo kodiranje v praksi srečamo redkeje. Vsaj v enem primeru je Huffmanov algoritem dokazano boljši od Shannon-Fanojevega; namreč,

če se frekvence simbolom povečujejo s potencami števila 2 (z drugimi besedami, če sledijo geometrijski porazdelitvi), Huffmanov algoritem sestavi **optimalne kode** [30].

$$\begin{array}{ll} \text{(a)} & \text{SF} = 15 \cdot 2 + 7 \cdot 2 + 6 \cdot 2 + 6 \cdot 3 + 5 \cdot 3 = 89 \;\; \text{bitov} \\ \text{(b)} & \text{HF} = 15 \cdot 1 + 7 \cdot 3 + 6 \cdot 3 + 6 \cdot 3 + 5 \cdot 3 = 87 \;\; \text{bitov} \end{array}$$

Primer 5.18: Izračun števila bitov za (a) Shannon-Fannojeve in (b) Huffmanove kode

Žal pa ima Huffmanovo kodiranje tudi pomembno slabost; postopek tako, kot ga je predstavil Huffman, ne zagotavlja, da bodo kode enolične. Ta primer je še posebej izrazit, ko imamo v seznamu prostih vozlišč več kot dva simbola z enakimi frekvencami. Težavo ponazorimo s primerom 5.19, kjer F hrani vrednosti frekvenc f_i, prirejenim simbolom $\sigma_i \in \Sigma$.

$$\begin{array}{l} \Sigma = \{\texttt{a, b, c, d, e}\} \\ F = [\,\texttt{4 \;\; 2 \;\; 2 \;\; 1 \;\; 1}] \end{array}$$

Primer 5.19: Abeceda sporočila s pripadajočim vektorjem frekvenc.

Del možnih združevanj prostih vozlišč vidimo na sliki 5.6. Vsaka od njih vodi do svojega drevesa $\mathcal{T}$. Zagotovo se najprej vprašamo, katero od teh dreves je najboljše. Ker so vsa drevesa Huffmanova, so vsa enako učinkovita, a s praktičnega stališča bi bilo zelo pomembno določiti, kako enolično tvoriti $\mathcal{T}$ in s tem dobiti tudi enolične Huffmanove kode. Pri tem nam pomaga nabor naslednjih zahtev:

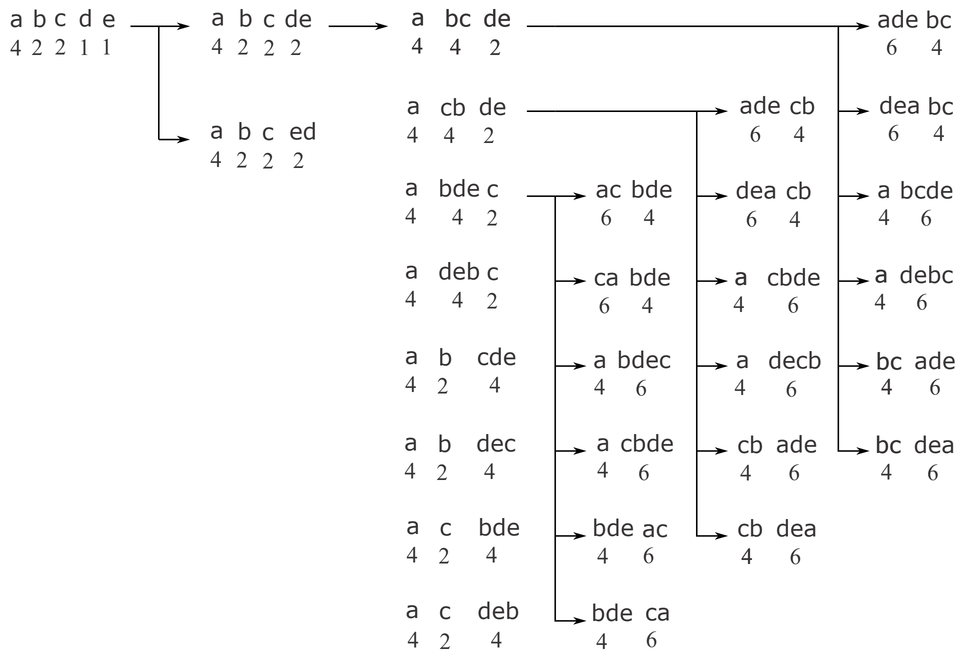
1. Prosta vozlišča naj bodo v seznamu prostih vozlišč vedno urejena glede na njihove frekvence v nenaraščajočem vrstnem redu. Če ima več prostih vozlišč enake frekvence, jih uredimo leksikografsko.

2. Prosti vozlišči združimo tako, da vozlišče, ki je na desni strani, z desne prilepimo k prostemu levemu vozlišču.

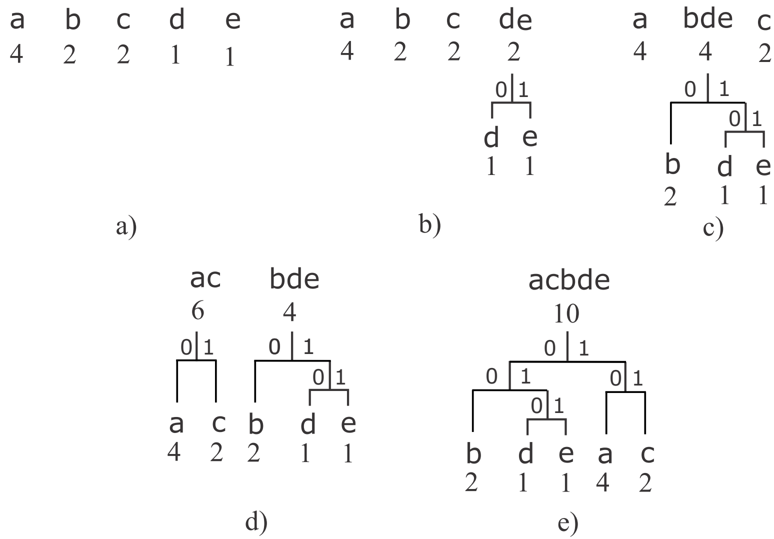
3. Levemu prostemu vozlišču dodelimo bit `0`, desnemu pa bit `1`.

4. Če je več prostih vozlišč z enako frekvenco, za združevanje izberemo vedno prosti vozlišči, ki sta najbolj oddaljeni.

Z upoštevanjem teh pravil, bi za primer 5.19 sestavili Huffmanovo drevo, ki ga vidimo na sliki 5.7.



Slika 5.6: Nekaj kombinacij prostih vozlišč



Slika 5.7: Postopek gradnje Huffmanovega drevesa z upoštevanjem pravil za enolično tvorbo Huffmanovega drevesa

Omenjen postopek zgradi $\mathcal{T}$ z **najmanjšo varianco kod** VLC. Varianca kode nam pove, za koliko se dolžina kode z najmanjšim številom bitov razlikuje od kode z največ biti.

Naj ob zaključku omenimo, da algoritmi stiskanja podatkov pogosto namesto frekvenc f_i simbolov σ_i uporabljajo njihove verjetnosti p_i, kar pa ne spremeni postopkov določanja kod VLC.

5.3.3 Huffmanov algoritem s prilagajanjem

Shannon-Fanojev in Huffmanov algoritem sta zagotovo imela ključno vlogo pri razvoju algoritmov za stiskanje podatkov. Huffmanov algoritem je pogost tudi v današnjih aplikacijah. A kritičen razmislek bi hitro našel tudi slabosti kot na primer:

1. Abecedo in tabelo frekvenc moramo zapisati v glavo datoteke, kar zmanjšuje učinkovitost stiskanja, predvsem pri krajših sporočilih.

2. Tabela frekvenc se ne prilagaja morebitni lokalni pogostosti pojavljanja simbolov (predstavlja globalno statistično značilnost sporočila).

3. Sporočilo, ki ga stiskamo, moramo poznati v celoti, da lahko določimo pogostost pojavljanja simbolov. To pa onemogoča uporabo v aplikacijah, kjer znaki prihajajo iz izvora znakov in bi jih morali takoj prenesti do sprejemnika znakov (tako imenovano **pretočno stiskanje**, ki je v času spletnih aplikacij nujno potrebno).

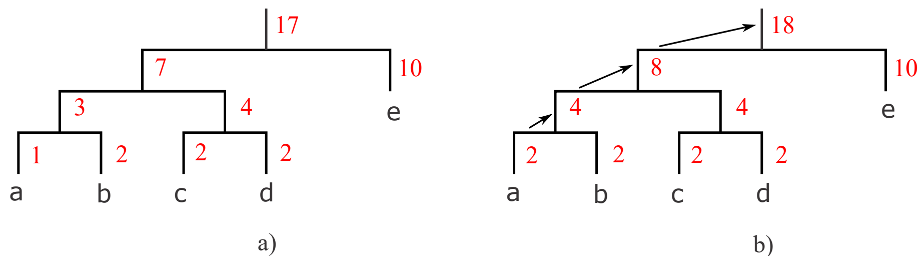
Omenjene tri slabosti lahko rešimo na dva načina:

1. Po vsakem prispelem simbolu σ_i ažuriramo njegovo frekvenco f_i in zgradimo novo Huffmanovo drevo $\mathcal{T}$.

2. Po vsakem prispelem simbolu σ_i ažuriramo njegovo frekvenco f_i in preverimo, ali je trenutno drevo $\mathcal{T}$ še Huffmanovo. Če ni, $\mathcal{T}$ spremenimo tako, da bo postalo Huffmanovo. Postopku pravimo **Huffmanov algoritem s prilagajanjem** (angl. adaptive Huffman algorithm), tudi **dinamični Huffmanov algoritem** (angl. dynamic Huffman algorithm).

Prvi način časovno zagotovo ni učinkovit in zato ni uporaben pri večjih sporočilih. Pri drugem načinu pa smo postavljeni pred nov izziv; kako ugotoviti, ali je drevo $\mathcal{T}$ po prispelem σ_i še Huffmanovo. Izkaže se, da je $\mathcal{T}$ Huffmanovo, če ima **lastnost dvojčkov** (angl. sibling property) [32].

Naj bo dvojiško drevo, ki ima vsako vozlišče uteženo. Teža w_i v listu drevesa je enaka frekvenci f_i simbola σ_i, ki se v tem listu nahaja. Teža notranjega vozlišča je vsota tež vozlišč neposrednih potomcev. Vsako vozlišče drevesa, ki ni koren, ima dvojčka. Dvojček je vozlišče, ki ima istega starša. Če obiščemo vozlišča drevesa z **obhodom po nivojih** (angl. level order), teže v vozliščih izpišemo v seznam L in je seznam urejen, potem ima drevo lastnost dvojčkov. Dvojiško drevo, ki ima lastnost dvojčkov, je Huffmanovo drevo.



Slika 5.8: (a) Trenutno Huffmanovo drevo, (b) razširjanje uteži (vrednosti uteži so označene z rdečo barvo)

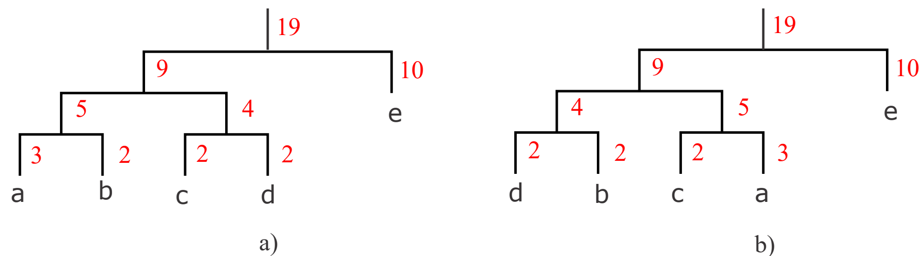
Na sliki 5.8a vidimo $\mathcal{T}$ s petimi znaki, kjer se je do zdaj že pojavil `a` enkrat, `b`, `c` in `d` dvakrat ter `e` desetkrat. Če uteži $\mathcal{T}$ izpišemo z obhodom po nivojih in uteži vstavimo v seznam L (primer 5.20a), $\mathcal{T}$ izpolnjuje lastnost dvojčkov, zato je $\mathcal{T}$ Huffmanovo drevo.

```
(a) L = ⟨1 2 2 2 3 4  7 10 17⟩
(b) L = ⟨3 2 2 2 5 4  9 10 19⟩
(c) L = ⟨2 2 2 5 4 7 11 10 21⟩
(d) L = ⟨2 2 2 4 5 6 11 10 21⟩
```

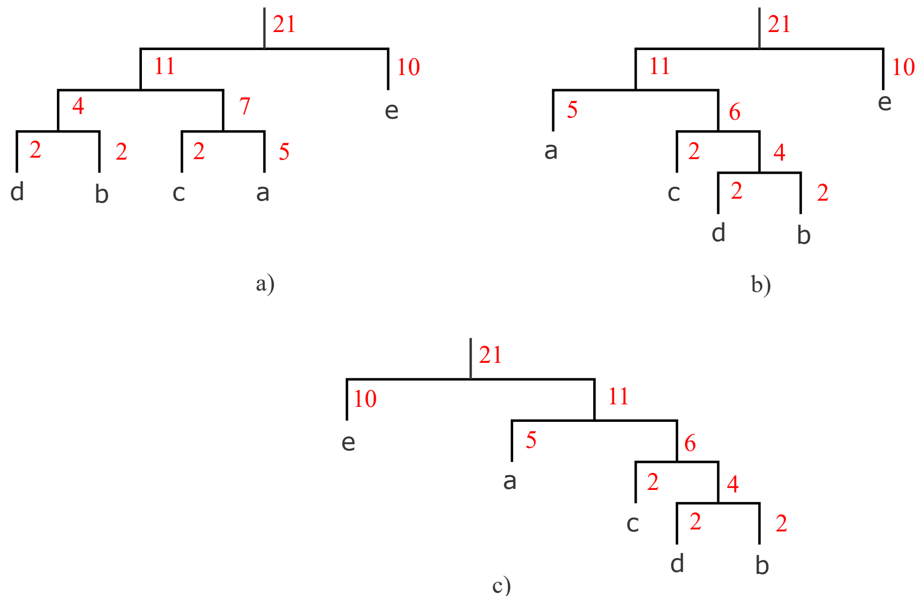
Primer 5.20: Stanje v seznamu L v različnih korakih algoritma

Predpostavimo, da je naslednji simbol v sporočilu simbol `a`. Najprej povečamo utež simbola za 1, nato pa ažuriramo stanje Huffmanovega drevesa. Opravimo tako imenovano **razširjanje uteži** (angl. propagation), od lista, v katerem je simbol, do korena drevesa. Med razširjanjem (označeno s puščicami na sliki 5.8b) inkrementiramo uteži vozlišč, na katere naletimo. Preverimo, ali je drevo Huffmanovo. Ker bi bil izpisan seznam uteži, tvorjen z obhodom po nivojih, urejen, lahko nadaljujemo, saj je $\mathcal{T}$ Huffmanovo. Naj bo tudi naslednji simbol `a`. Po opravljenem razširjanju uteži dobimo drevo na sliki 5.9a. Če po pravilu izpišemo uteži, dobimo neurejen seznam

(glej primer 5.20b). Lastnost dvojčkov je prekršena, drevo ni več Huffmanovo, zato ga moramo spremeniti. Prvo neurejenost odpravimo tako, da zamenjamo skrajno vozlišče, ki ima utež 2 z vozliščem, ki ima utež 3; zamenjati moramo vozlišči a in d. Po njuni zamenjavi ažuriramo tudi vrednosti uteži v notranjih vozliščih. Dobimo $\mathcal{T}$ na sliki 5.9b, čigar L je urejen, drevo izpolnjuje lastnost dvojčkov, torej je Huffmanovo.



Slika 5.9: (a) Razširjanje uteži in (b) zamenjava vozlišč



Slika 5.10: (a) Huffmanovo drevo po prihodu dveh dodatnih simbolov a, (b) prva zamenjava vozlišč in (c) druga zamenjava vozlišč

Naj pride simbol a še dvakrat, tako da je njegova utež že 5. Po razširjanju uteži dobimo stanje, kot ga vidimo na sliki 5.10a. Če izpišemo uteži, dobimo v seznamu L situacijo, ki jo kaže primer 5.20c. Najprej zamenjamo vozlišči

z utežema 4 in 5 in popravimo uteži v notranjih vozliščih. Dobimo drevo, ki ga vidimo na sliki 5.10b. Ponovno izpišemo uteži z obhodom po nivojih. V L dobimo stanje iz primera 5.20d. Opraviti moramo zamenjavo, tokrat vozlišč z utežjo 11 in 10 tako, da dobimo drevo na sliki 5.10c, ki je Huffmanovo, saj je seznam L urejen.

Vidimo, da so se kode simbolov glede na začetno stanje, bistveno spremenile. Simbol e smo prej kodirali z bitom 1, zdaj z bitom 0. Simbol a je bil najprej predstavljen s tremi biti $\langle$000$\rangle$, ko je narasla njegova utež, pa je kodiran s samo dvema bitoma, to je $\langle$10$\rangle$. Simbol c je še vedno kodiran s tremi biti, a so se njegove kode spremenile. Vozlišči b in d sta nazadovali in sta zdaj kodirani s štirimi biti.

Pri Huffmanovem algoritmu s prilagajanjem pa nastopi še ena težava. Vnaprej praviloma ne poznamo abecede Σ, iz katerega je sestavljeno sporočilo I. Najpreprostejša rešitev te težave je inicializacija Huffmanovega drevesa z vsemi možnimi znaki (lahko bi uporabili vseh 256 znakov ASCII) in postavitev njihovih začetnih vrednosti frekvenc $f_i = 0$. Znaki, ki se bodo pojavljali pogosteje, bodo postopoma začeli prejemati krajše kode. Vsaj na začetku in pri krajših sporočilih metoda ni učinkovita. Boljša rešitev je, da začnemo kodirni proces s praznim drevesom in dodajamo simbole samo, ko jih preberemo iz sporočila. A pred tem moramo odpraviti še eno težavo; kako zakodirati simbol, ki še ni član $\mathcal{T}$? Rešitev je uporaba ubežne kode, ESC, s katero bomo preklapljali med kodiranjem simbolov σ_i s Huffmanovimi ali s kodami ASCII. Ko bo dekodirnik zaznal kodo ESC, bo vedel, da je naslednji simbol zakodiran z 8-bitno ASCII-kodo, sicer pa bo koda Huffmanova. Ta simbol bomo nato vgradili v $\mathcal{T}$ in poskrbeli, da bo izpolnjevalo lastnost dvojčkov. $\mathcal{T}$ inicializiramo z dvema simboloma: z EOF in ESC, ki imata na začetku utež 0. Pomen ubežne kode ESC že poznamo, kodo EOF (angl. end of file) pa bomo zapisali v izhodno zaporedje O po zaključku kodiranja. S tem bomo dekodirniku povedali, naj zaključi razširjanje. Kodirnik/dekodirnik bi lahko implementirali tudi drugače, in sicer tako, da bi namesto znaka EOF uvedli glavo stisnjene datoteke, v katero bi zapisali število znakov v sporočilu I. Žal pa na ta način ne moremo podpreti pretočnega stiskanja, pri katerem vnaprej ne vemo, koliko znakov je treba stisniti.

V nadaljevanju si oglejmo delovanje Huffmanovega algoritma s prilagajanjem za zaporedje iz primera 5.21.

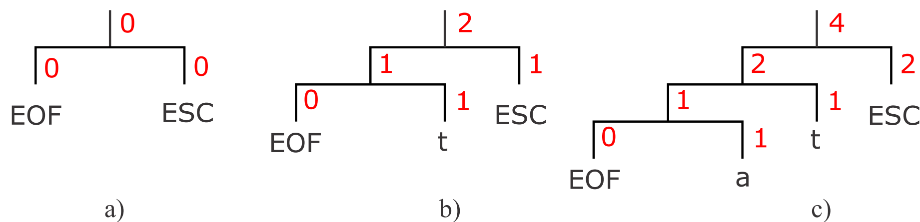
$$I = \langle \texttt{tata}\langle EOF\rangle\rangle$$

Primer 5.21: Huffman s prilagajanjem – primer

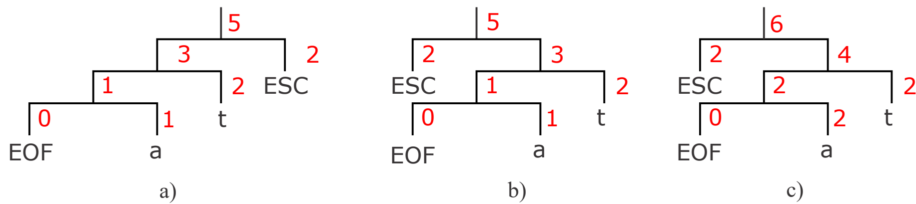
Inicializacijo drevesa $\mathcal{T}$ kaže slika 5.11. $\mathcal{T}$ izpolnjuje lastnost dvojčkov in je zato Huffmanovo. Ugodno je, da je EOF v levi veji. Prvi simbol t še ni član Huffmanovega drevesa, zato kodirnik pošlje ubežni znak, kodiran z bitom 1, ki mu sledi 8 bitov iz tabele ASCII, ki kodirajo simbol t $= \langle$01110100$\rangle$ (v nadaljevanju bomo zaradi preglednosti namesto bitov ASCII zapisali kar simbol). t nato vgradimo v $\mathcal{T}$. Nov simbol vedno vstavimo desno od EOF. Vstavljenemu simbolu dodelimo utež 1, utež ESC inkrementiramo, nato pa opravimo razširjanje uteži (stanje vidimo na sliki 5.11b). Preverimo, ali $\mathcal{T}$ izpolni lastnosti dvojčkov. Vidimo, da so uteži v vozliščih urejene, če jih izpišemo po nivojih, zato je $\mathcal{T}$ Huffmanovo. Zato lahko nadaljujemo s kodiranjem. Tudi naslednjega simbola $\sigma_i =$ a še ni v Huffmanovem drevesu, zato pošljemo na izhod ubežno kodo (tudi tokrat je to bit 1), a pa vstavimo v $\mathcal{T}$. Novemu simbolu priredimo utež 1 (saj smo ga v I srečali prvič), inkrementiramo utež ESC (do zdaj smo ESC na izhod poslali dvakrat) ter poženemo razširjanje uteži. Dobimo situacijo na sliki 5.11c. Hitro ugotovimo, da je tudi tokrat $\mathcal{T}$ Huffmanovo, zato kodiramo naslednji simbol. Simbol t je že v drevesu, zato pošljemo na izhod njegovo Huffmanovo kodo $\langle$01$\rangle$, inkrementiramo njegovo utež in sprožimo razširjanje uteži. Dobimo $\mathcal{T}$, ki ga vidimo na sliki 5.12a. $\mathcal{T}$ tokrat ni Huffmanovo, v Huffmanovo ga spremenimo z zamenjavo vozlišč (slika 5.12b). Naslednji simbol sporočila je a, ki je že v drevesu, zato pošljemo na izhod njegovo kodo $\langle$101$\rangle$. Povečamo njegovo utež in opravimo razširjanje uteži. Ugotovimo, da je drevo Huffmanovo. Ker smo prišli do konca sporočila, kodirnik na izhod pošlje še kodo za konec sporočila EOF $= \langle$100$\rangle$ in kodiranje zaključi. Rezultat kaže primer 5.22.

(a) $O = \langle$1t1a01101100$\rangle$
(b) $O = \langle$1011101001011000010110110 0$\rangle$

Primer 5.22: Izhodno bitno zaporedje, kjer so simboli predstavljeni (a) s simboli in (b) z bitnimi kodami ASCII.



Slika 5.11: Huffmanovo kodiranje s prilagajanjem; primer – prvi del



Slika 5.12: Huffmanovo kodiranje s prilagajanjem; primer – drugi del

Dekodiranje/razširjanje poteka zelo podobno kot samo stiskanje. Dekodirnik inicializira $\mathcal{T}$ na enak način kot kodirnik (slika 5.11a). Vhod v dekodirnik je izhod iz dekodirnika; to je bitno zaporedje iz primera 5.22. Dekodirnik se postavi v koren $\mathcal{T}$ in prebere prvi bit, to je bit 1, ki predstavlja ukaz za premik v desno vejo $\mathcal{T}$. Ugotovimo, da smo pristali v listu, v katerem najdemo simbol ESC, kar pomeni, da moramo prebrati 8 bitov in iz tabele ASCII določiti znak. Na ta način dekodiramo simbol t, ki ga vpišemo v izhodno zaporedje $O = \langle \mathsf{t} \rangle$. Simbol t nato vstavimo v drevo, in sicer v levo vejo desno od znaka EOF. Simbolu t damo utež 1, simbolu ESC pa njegovo utež inkrementiramo ter opravimo razširjanje uteži. Dobimo $\mathcal{T}$, ki ga vidimo na sliki 5.11b. Nato preverimo, če $\mathcal{T}$ ugodi lastnosti dvojčkov. Ugotovimo, da lastnost dvojčkov velja, zato se postavimo na vrh drevesa in preberemo naslednji bit iz I. Tudi tokrat je to bit 1, ki usmeri dekodirnik v desno vejo, kjer je ESC. Preberemo 8 bitov in s pomočjo tabele ASCII dekodiramo a. Znak pošljemo v O, $O = \langle \mathsf{ta} \rangle$, nato pa ga vstavimo v $\mathcal{T}$ z utežjo 1, inkrementiramo utež simbola ESC, opravimo razširjanje uteži ter preverimo lastnost dvojčkov. Ta velja tudi tokrat ($\mathcal{T}$ vidimo na sliki 5.11c), zato se postavimo na vrh trenutnega $\mathcal{T}$ ter preberemo naslednji bit. Bit ima vrednost 0, zato se usmerimo po levi veji do notranjega vozlišča $\mathcal{T}$, kar pomeni, da se branje bitov lahko nadaljuje. Naslednji bit je 1, zato po desni veji $\mathcal{T}$ dosežemo list, v katerem je t. t pošljemo na izhod $O = \langle \mathsf{tat} \rangle$, simbolu t pa v $\mathcal{T}$ povečamo utež ter opravimo razširjanje uteži (slika 5.12a). Ugotovimo, da pravilo dvojčkov več ne velja, zato preuredimo $\mathcal{T}$. Dobimo drevo, ki ga vidimo na sliki 5.12b. Dekodirnik se nato postavi na vrh $\mathcal{T}$ ter prebere naslednji bit iz I. Ta bit je 1, zato se premakne po desni veji. Ugotovi, da je vozlišče notranje, zato prebere naslednji bit, to je bit 0. Po premiku po levi veji pridemo v naslednje notranje vozlišče. Kodirnik zato prebere naslednji bit, to je bit 1. Ta algoritem usmeri v desno vejo, kjer doseže list s simbolom a. Simbol pošlje na izhod O, $O = \langle \mathsf{tata} \rangle$, poveča utež v listu s simbolom a, opravi razširjanje uteži ter preveri lastnost dvojčkov. Ker ta velja, se algoritem ponovno postavi v koren drevesa, prebere bite $\langle 100 \rangle$ ter ugotovi, da je v doseženem listu EOF, zato dekodiranje zaključi.

5.3.4 Poenostavljen algoritem stiskanja s prilagajanjem

V tem podpoglavju si bomo pogledali zelo preprosto različico algoritma stiskanja podatkov s prilagajanjem, ki je manj učinkovita, kot je Huffmanov algoritem s prilagajanjem, saj kode, prirejene simbolom, niso Huffmanove, vsaj v splošnem ne. Predpostavimo, da imamo sporočilo $I = \langle \sigma_i \rangle$, $\sigma_i \in \Sigma$. Na začetku najprej tvorimo $|\Sigma|$ **predponskih kod** in jih priredimo simbolom σ_i. Kode in simbole vstavimo v seznam L treh komponent: simbol σ_i, njegova frekvenca f_i in priponska koda, kjer krajše priponske kode postavimo na začetek seznama. Kodiranje poteka na naslednji način. Predpostavimo, da kodiramo simbol σ_i. Najprej ga poiščemo v L ter na izhod pošljemo njegovo prefiksno kodo. Nato mu povečamo njegovo frekvenco f_i. Preverimo, ali so simboli urejeni glede na f_i. Če niso, poiščemo v L prvi simbol z večjo frekvenco od frekvence f_i in simbola (in njune frekvence) zamenjamo, položajev prefiksnih kod pa pri tem ne spreminjamo. Oglejmo si primer 5.23.

$$I = \langle \texttt{cdd} \rangle$$
$$\Sigma = \{\texttt{a,b,c,d,e}\}$$

Primer 5.23: Prirejen Huffmanov algoritem – primer

Začetno stanje kodirnika kaže primer 5.24a. Prvi simbol v I je c. Na izhod pošljemo $O = \langle 110 \rangle$, nato simbolu c inkrementiramo frekvenco. Ker L ni več urejen glede na frekvence, zamenjamo simbola a in c (glej primer 5.24b). Naslednji simbol je d, katerega kodo prilepimo k izhodnemu zaporedju bitov (torej $O = \langle 1101110 \rangle$), nato pa simbol d nagradimo s krajšo kodo (zamenjamo ga s simbolom b, kot kaže primer 5.24c). Simbol d se pojavi še enkrat. Njegova frekvenca $f = 2$, zato zamenja mesto s simbolom c, izhodno zaporedje pa je $O = \langle 110111010 \rangle$. Končno stanje prikazuje primer 5.24d. Dekodiranje poteka na enak način, le da postopke dekodirnika nadziramo z branjem bitov.

Opisano metodo enostavno razširimo s simboloma ESC in EOF tako, da nam ni treba vnaprej vedeti, iz katerih simbolov sestoji I.

σ_i	f_i	koda	σ_i	f_i	koda
a	0	0	c	1	0
b	0	10	b	0	10
c	0	110	a	0	110
d	0	1110	d	0	1110
e	0	1111	e	0	1111
	a)			b)	

σ_i	f_i	koda	σ_i	f_i	koda
c	1	0	d	2	0
d	1	10	c	1	10
a	0	110	a	0	110
b	0	1110	b	0	1110
e	0	1111	e	0	1111
	c)			d)	

Primer 5.24: Prirejanje kod pri algoritmu s prilagajanjem

5.4 Aritmetično kodiranje

Huffmanovo kodiranje je, kot smo že omenili, optimalno, ko je frekvenca pojavljanja simbolov v sporočilu enaka potenci števila 2. Takšna porazdelitev simbolov je v realnih razmerah redkokdaj izpolnjena. Posledica je manj učinkovito stiskanje, kot je teoretično možno, to je, kot določa Shannonova entropija (enačba 5.4). Če je verjetnost znaka $p(\sigma_i) = \frac{1}{3}$, bi optimalno število bitov za kodiranje σ_i bilo $-\log_2\left(\frac{1}{3}\right) \approx 1{,}585$ bita, s Huffmanovim kodiranjem pa bi σ_i morali kodirati z dvema bitoma. Na žalost je ta lastnost Huffmanovega algoritma še najbolj očitna pri simbolih z veliko verjetnostjo, torej pri simbolih, ki se velikokrat ponavljajo in je izguba pri njihovem kodiranju še posebej boleča. Če je $p(\sigma_i) = 0{,}9$, bi bila optimalna koda dolga $-\log_2 0{,}9 \approx 0{,}152$ bita. Huffmanov algoritem bi σ_i seveda priredil 1-bitno kodo, ki pa je več kot šestkrat daljša, kot bi to bilo treba. Toda, kako naj simbolu σ_i priredimo manj kot en bit? Da bi to lahko storili, moramo najprej izstopiti iz diskretnega sveta digitalnega računalnika. Že v šestdesetih letih prejšnjega stoletja so, neobremenjeni z digitalno tehniko, izumili nov način kodiranja, ki je to zmogel; to je bilo **aritmetično kodiranje** (angl. arithmetic coding). Odlične opise teoretičnih in praktičnih vidikov aritmetičnega kodiranja najdemo v [37, 38, 30].

5.4.1 Ideja aritmetičnega kodiranja

Ideja aritmetičnega kodiranja je v bistvu zelo preprosta. Razmišljajmo takole: Prav gotovo je število različnih sporočil, ki bi jih morebiti morali stisniti, veliko. Tako veliko, da lahko rečemo, da jih je neskončno. Prav tako vemo, da je na poljubnem številskem intervalu tudi neskončno števil. Morda bi lahko sporočilo I zakodirali tako, da bi mu enolično priredili število v iz danega intervala. Če bi v bilo sestavljeno iz malo števk, bi I morebiti stisnili. Oglejmo si primer zaporedja I iz primera 5.25. Simbolom σ_i najprej

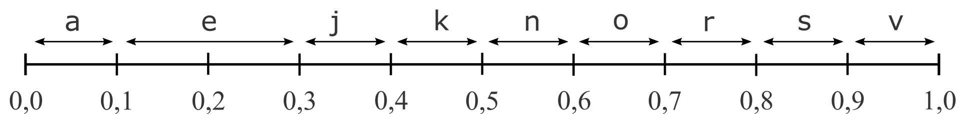
(a) $I = \langle$`kresovanje`$\rangle$
(b) $\Sigma = \{$`a e j k n o r s v`$\}$
(c) $P = [0{,}1 \quad 0{,}2 \quad 0{,}1 \quad 0{,}1 \quad 0{,}1 \quad 0{,}1 \quad 0{,}1 \quad 0{,}1 \quad 0{,}1]$

Primer 5.25: (a) Vhodno zaporedje, (b) njegova abeceda in (c) vektor verjetnosti

določimo njihovo verjetnost pojavljanja v I; dobimo vektor verjetnosti P (glej primer 5.25c). Simbole σ_i nato razporedimo na interval $\mathcal{I} = [0{,}0,\ 1{,}0)$ tako, da dolžina podintervala, ki ga zaseda σ_i, ustreza njegovi verjetnosti

$p(\sigma_i)$, to je, verjetnejši simboli dobijo daljši interval.

Vrstni red simbolov na $\mathcal{I}$ je v principu poljuben, pomembno je le, da kodirnik in dekodirnik uporabljata isti vrstni red. Tega najlažje zagotovimo tako, da uporabimo kar abecedni vrstni red. Rezultat opisanega postopka za sporočilo I vidimo na sliki 5.13 oziroma v razpredelnici 5.7. Simbol $\sigma_1 =$ e si je zagotovil večji interval, saj se edini pojavi dvakrat.



Slika 5.13: Simbolom sporočila I priredimo podintervale na intervalu $\mathcal{I}$

Razpredelnica 5.7: Simboli, njihova verjetnost ter položaj na intervalu $\mathcal{I}$

σ_i	$p(\sigma_i)$	območje na $\mathcal{I}$
a	0,1	[0,0, 0,1)
e	0,2	[0,1, 0,3)
j	0,1	[0,3, 0,4)
k	0,1	[0,4, 0,5)
n	0,1	[0,5, 0,6)
o	0,1	[0,6, 0,7)
r	0,1	[0,7, 0,8)
s	0,1	[0,8, 0,9)
v	0,1	[0,9, 1,0)

Proces, ki bo enolično priredil sporočilu I vrednost $v \in \mathcal{I}$, bomo opravili z enačbo 5.5,

$$\begin{aligned} mLow &= \quad mLow + (mHigh - mLow)\ mLow(\sigma_i) \\ mHigh &= \quad mLow + (mHigh - mLow)\ mHigh(\sigma_i) \end{aligned} \tag{5.5}$$

kjer so:

- $[mLow, mHigh)$: spodnja in zgornja meja trenutnega intervala (imenovali ga bomo tudi **delovni interval**); začetni vrednosti sta določeni z mejami intervala $\mathcal{I}$;

- $mLow(\sigma_i)$: spodnja meja simbola σ_i na intervalu $\mathcal{I}$ in

- $mHigh(\sigma_i)$: zgornja meja simbola σ_i na intervalu $\mathcal{I}$.

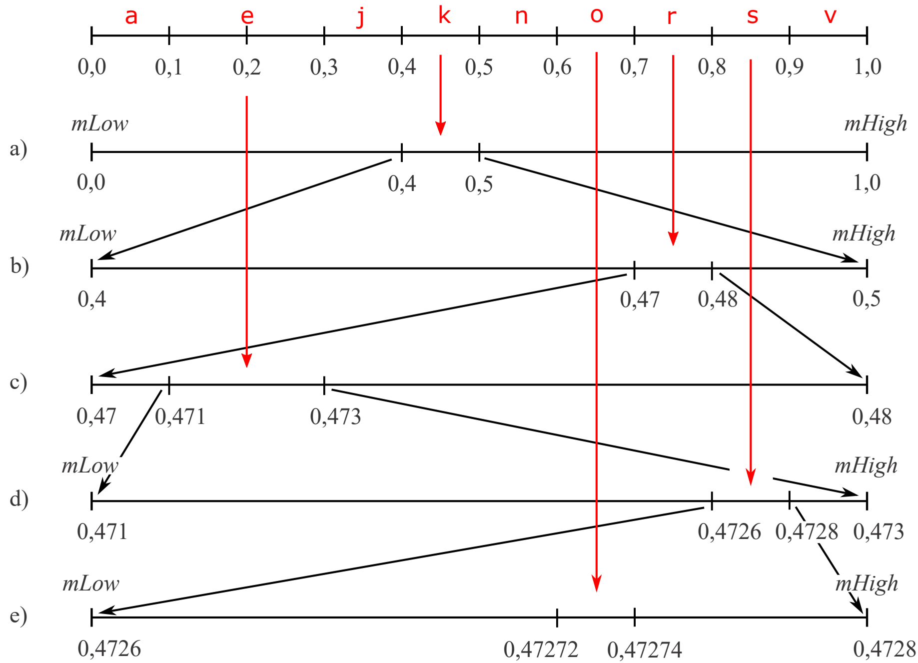
Algoritem po inicializaciji spremenljivk $mLow$ in $mHigh$ po vrsti bere simbole $\sigma_i \in I$, $0 \leq i < |I| - 1$. Za vsakega izračuna nove vrednosti delovnega intervala $[mHigh, mLow)$. Postopek za naše sporočilo vidimo v razpredelnici 5.8.

Razpredelnica 5.8: Proces aritmetičnega kodiranja

σ_i	$mLow$	$mHigh$
/	0,0	1,0
k	0,4	0,5
r	0,47	0,48
e	0,471	0,473
s	0,4726	0,4728
o	0,47272	0,47274
v	0,472738	0,47274
a	0,472738	0,4727382
n	0,4727381	0,47273812
j	0,472738106	0,472738108
e	0,4727381062	0,4727381066

Prvi znak sporočila $\sigma_0 =$ k zoži delovni interval na [0,40, 0,50). Vsako vrednost v, ki bo znotraj tega intervala, bomo, posledično, med razširjanjem interpretirali kot simbol k. Naslednji znak $\sigma_1 =$ r, ki na $\mathcal{I}$ zaseda podinterval [0,70, 0,80), nadalje skrči delovni interval na [0,47, 0,48). Vsako število v iz tega podintervala predstavlja sporočilo ⟨kr⟩. Postopek krčenja delovnega intervala za prve štiri simbole sporočila I shematično vidimo tudi na sliki 5.14.

Sporočilo $I =$ ⟨kresovanje⟩ predstavlja katerokoli število v iz zadnjega delovnega intervala, to je [0,4727381062, 0,4727381066). Seveda je smiselno, da izberemo takšen v, ki ima najmanj števk. V dobljenem intervalu pa žal takšne vrednosti ni, še najkrajša je spodnja vrednost $mLow$, zato izberemo kar to, $v =$ 0,4727381062. Spoprimimo se še s postopkom **dekodiranja/razširjanja**. Ta potrebuje podatke o stanju na intervalu $\mathcal{I}$, nato pa v vsakem koraku najprej ugotovi, v katerem podintervalu intervala $\mathcal{I}$ je vrednost v, ter izpiše na njem nahajajoč se simbol σ_i. Algoritem nato izračuna novo vrednost v z enačbo 5.6. Potek razširjanja za naš primer vidimo v razpredelnici 5.9.



Slika 5.14: Nekaj korakov krčenja delovnega intervala $[mLow,\ mHigh)$

$$v = \frac{v - mLow(\sigma_i)}{mHigh(\sigma_i) - mLow(\sigma_i)} \tag{5.6}$$

Oglejmo si še primer 5.26. Simbol $\sigma_0 =$ `a` naj zaseda interval [0,0, 0,9), $\sigma_1 =$ `!` pa [0,9, 1,0). Postopek aritmetičnega kodiranja nam kaže razpre-

(a) $I = \langle$`aaaaaaaaa!`$\rangle$
(b) $\Sigma = \{$`a !`$\}$
(c) $P = [0{,}9 \quad 0{,}1]$

Primer 5.26: (a) Vhodno zaporedje, sestavljeno iz dveh simbolov, (b) abeceda in (c) vektor verjetnosti

delnica 5.10. Tokrat je končni delovni interval dovolj širok, da na njem najdemo tudi vrednosti z malo števkami. Na primer, vrednosti 0,35, 0,36, 0,37 in 0,38 imajo le dve decimalki, te vrednosti pa lahko izberemo za razširjanje sporočila.

Razpredelnica 5.9: Postopek aritmetičnega dekodiranja

v	σ_i	$mLow(\sigma_i)$	$mHigh(\sigma_i)$
0,4727381062	k	0,4	0,5
0,727381062	r	0,7	0,8
0,27381062	e	0,1	0,3
0,8690531	s	0,8	0,9
0,690531	o	0,6	0,7
0,90531	v	0,9	1,0
0,0531	a	0,0	0,1
0,531	n	0,5	0,6
0,31	j	0,3	0,4
0,1	e	0,1	0,3
0,0	/	/	/

Razpredelnica 5.10: Kodiranje sporočila $I = \langle$aaaaaaaaa!$\rangle$

σ_i	$mLow$	$mHigh$
/	0,0	1,0
a	0,0	0,9
a	0,0	0,81
a	0,0	0,729
a	0,0	0,6561
a	0,0	0,59049
a	0,0	0,531441
a	0,0	0,4782969
a	0,0	0,43046721
a	0,0	0,387420489
!	0,34867844010	0,3874204890

5.4.2 Celoštevilske implementacije aritmetičnega kodiranja

Pri realizaciji ideje aritmetičnega kodiranja, predstavljene v podpoglavju 5.4.1, bi se zelo hitro zapletlo. Digitalni računalnik namreč deluje v **končni aritmetiki** in ne more predstaviti vseh števil (točno lahko predstavi samo nekatera števila, ki jim pravimo **predstavljiva števila**), pri aritmetičnem kodiranju pa smo predpostavili, da je na intervalu $\mathcal{I}$ neskončno

števil. Zaradi končnega števila predstavljivih števil se bosta v nekem trenutku vrednosti delovnega intervala $mLow$ in $mHigh$ zlili in kodiranje se ne bo uspešno zaključilo. Zato je bilo treba najti drugačne možnosti implementacije, ki bodo delovale s končno aritmetiko, najbolje kar s celoštevilsko. V nadaljevanju si bomo ogledali dve takšni rešitvi: implementacijo s pomikanjem [32] in implementacijo s skaliranjem [37].

5.4.2.1 Implementacija aritmetičnega kodiranja s pomikanjem

Idejo aritmetičnega kodiranja s pomikanjem bomo hitro razumeli, če si še enkrat ogledamo razpredelnico 5.8. Vidimo, da se najbolj pomembni števki v $mLow$ in $mHigh$ več ne spremenita, ko postaneta enaki. Zato te števke pravzaprav ne bi bilo več treba hraniti v registrih procesorja, ampak bi jo lahko *potisnili* na izhod. S tem bi pridobili mesto za novo števko v spremenljivkah delovnega intervala $mLow$ in $mHigh$. Prav to idejo uporablja aritmetično kodiranje s pomikanjem.

Oglejmo si, kako bi kodirali sporočilo $I = \langle \textsf{kresovanje} \rangle$, za katerega že poznamo razporeditev simbolov σ_i na intervalu $\mathcal{I}$ (glej sliko 5.13). A tokrat bomo začetno vrednost delovnega intervala inicializirali nekoliko drugače. Za lažjo razlago naj bo $[mLow = 0, mHigh = 99999)$. V dejanski implementaciji bi $mHigh$ hranil največjo vrednost, ki jo lahko shranimo v spremenljivki tipa *unsigned int*. Tudi enačbo za določitev nove vrednosti delovnega intervala bomo nekoliko spremenili (enačba 5.7):

$$
\begin{aligned}
mLow &= mLow + (mHigh - mLow + 1)\ mLow(\sigma_i) \\
mHigh &= mLow + (mHigh - mLow + 1)\ mHigh(\sigma_i) - 1
\end{aligned}
\tag{5.7}
$$

Aritmetično kodiranje vsakokrat, ko sta najbolj pomembni števki v $mLow$ in $mHigh$ enaki, to števko pošlje na izhod, preostale števke v $mLow$ in $mHigh$ pa premakne v levo. Na najmanj pomembno izpraznjeno v mesto v $mLow$ vrine števko 0, v $mHigh$ pa 9. Če najpomembnejši števki nista enaki, ne opravimo premika in na izhod ne pošljemo ničesar. Ko zakodiramo vse simbole sporočila, pošljemo na izhod še dve najbolj pomembni števki iz $mLow$ (ali $mHigh$), ki nam bosta omogočili sporočilo tudi v celoti dekodirati. Postopek aritmetičnega kodiranja s pomikanjem kaže razpredelnica 5.11. Vidimo, da je pridobljena vrednost (primer 5.27) popolnoma skladna z vrednostjo, ki smo jo dobili z idealnim aritmetičnim kodiranjem, razlikuje se le v zadnjih dveh števkah. Prav tako smo opazili, da kodirnik pri simbolu $\textsf{e}$, ki ima največjo verjetnost pojavljanja, na izhod ni poslal nobenega znaka, kar je za stiskanje podatkov odlična novica.

Postopek dekodiranja (vidimo ga v razpredelnici 5.12) je naslednji:

Razpredelnica 5.11: Postopek aritmetičnega kodiranja s pomikanjem

	$mLow$	$mHigh$	premik	izhod
inicializacija	00000	99999		
kodiramo `k`	40000	49999	DA	4
stanje po premiku	00000	99999		
kodiramo `r`	70000	79999	DA	7
stanje po premiku	00000	99999		
kodiramo `e`	10000	29999	NE	
kodiramo `s`	26000	27999	DA	2
stanje po premiku	60000	79999		
kodiramo `o`	72000	73999	DA	7
stanje po premiku	20000	39999		
kodiramo `v`	38000	39999	DA	3
stanje po premiku	80000	99999		
kodiramo `a`	80000	81999	DA	8
stanje po premiku	00000	19999		
kodiramo `n`	10000	11999	DA	1
stanje po premiku	00000	19999		
kodiramo `j`	06000	07999	DA	0
stanje po premiku	60000	79999		
kodiramo `e`	62000	65999	DA	6
stanje po premiku	20000	59999		
dve števki iz $mLow$				20

$$O = \langle \texttt{47273810620} \rangle$$

Primer 5.27: Izhod aritmetičnega kodiranja s pomikanjem

1. Izhod kodirnika postane vhod I v dekodirnik.

2. Poznati moramo abecedo Σ in verjetnosti $p(\sigma_i)$ simbolov $\sigma_i \in \Sigma$.

3. Inicializiramo $mLow$ in $mHigh$ na enak način, kot ju je inicializiral kodirnik.

4. Včitamo ustrezno število števk v register/spremenljivko R.

5. Opravimo dekodirni postopek, v katerem v vsakem koraku

- določimo vrednost v glede na števke v R in delovni interval $[mLow, mHigh)$ z enačbo 5.8

$$v = \frac{R - mLow}{mHigh - mLow + 1}, \tag{5.8}$$

- posodobimo delovni interval $[mLow, mHigh)$ z enačbo 5.7,
- če sta najpomembnejši števki v $mLow$ in $mHigh$ enaki, opravimo pomike v $mLow$, $mHigh$ in R; v R na položaj najmanj pomembne števke vstavimo naslednjo števko iz I.

6. Z dekodiranjem zaključimo, ko smo v R vrinili $|R| - 2$ dodatnih ničel.

Razpredelnica 5.12: Postopek dekodiranja s pomikanjem

	R	v	znak	$mLow$	$mHigh$	pomik
inicializacija	47273			00000	99999	
dekodiranje		0,47273	k	40000	49999	DA
pomik	72738			00000	99999	
dekodiranje		0,72738	r	70000	79999	DA
pomik	27381			00000	99999	
dekodiranje		0,27381	e	10000	29999	NE
dekodiranje		0,86905	s	26000	27999	DA
pomik	73810			60000	79999	
dekodiranje		0,69050	o	72000	73999	DA
pomik	38106			20000	39999	
dekodiranje		0,90530	v	38000	39999	DA
pomik	81062			80000	99999	
dekodiranje		0,05310	a	80000	81999	DA
pomik	10620			00000	19999	
dekodiranje		0,53100	n	10000	11999	DA
pomik	06200			00000	19999	
dekodiranje		0,31000	j	06000	07999	DA
pomik	62000			60000	79999	
dekodiranje		0,10000	e	62000	65999	DA
pomik	20000			20000	59999	

S primerjavo razpredelnic 5.11 in 5.12 vidimo, kako skladno delujeta kodirnik in dekodirnik, razen na koncu, ko smo poslali na izhod dve zadnji

števki iz $mLow$, s katerima omogočimo dekodiranje zadnjih treh simbolov. Zadnji dve števki tudi ostaneta kot najpomembnejši števki v R po zaključku dekodiranja (v razpredelnici 5.12 smo ju označili z rdečo).

V razpredelnici 5.13 pokažemo še postopek celoštevilskega kodiranja s pomikanjem za sporočilo iz primera 5.26. Vidimo, da kodirnik sploh ne pošlje znaka na izhod med kodiranjem simbola a, to je simbola z veliko verjetnostjo. Rezultat, ki ga dobimo s postopkom kodiranja s pomikanjem je enak tistemu, ki smo ga dobili z izvornim postopkom, kjer $mLow$ in $mHigh$ predstavimo s plavajočo vejico.

Razpredelnica 5.13: Postopek dekodiranja s pomikanjem v primeru sporočila, ko je verjetnost pojava enega simbola velika

	$mLow$	$mHigh$	premik	izhod
inicializacija	00000	99999		
kodiramo a	00000	89999	NE	
kodiramo a	00000	80999	NE	
kodiramo a	00000	72899	NE	
kodiramo a	00000	65609	NE	
kodiramo a	00000	59048	NE	
kodiramo a	00000	53143	NE	
kodiramo a	00000	47828	NE	
kodiramo a	00000	43045	NE	
kodiramo a	00000	38740	NE	
kodiramo EOF	34866	38740	DA	3
stanje po premiku	48660	87409		
dve števki iz $mLow$				348

Primeri, ki smo jih prikazali do zdaj, so zagotovo korak v pravo smer, a na zelo pomembno podrobnost morda niti nismo bili pozorni. V vsaki iteraciji kodiranja se je delovni interval $[mLow, mHigh)$, ko nismo poslali simbola na izhod, skrčil. Tudi v celoštevilski implementaciji bi se zato lahko znašli v pasti, ko bi se meji $mLow$ in $mHigh$ tako približali (v skrajnosti bi se celo izenačili), da bi nadaljnje kodiranje postalo nemogoče. Iz zagate se rešimo z naslednjim postopkom [32]:

1. Uvedemo števec podkoračitve u in ga postavimo na vrednost 0.

2. Če se najpomembnejši števki v $mLow$ in $mHigh$ razlikujeta za 1, preverimo drugi najpomembnejši števki.

3. Če je druga najpomembnejša števka pri $mHigh$ 0, pri $mLow$ pa 9, je delovni interval preozek in moramo ukrepati.

4. Odstranimo drugo najpomembnejšo števko (0 iz $mLow$ in 9 iz $mHigh$), preostale manj pomembne števke pa premaknemo za eno mesto v levo.

5. Na položaj najmanj pomembne števke v $mHigh$ vstavimo 9 in 0 v $mLow$.

6. Inkrementiramo u.

7. Postopek ponavljamo od koraka 2 tako dolgo, dokler nevarnosti podkoračitve ne odpravimo (dovolj razmaknemo delovni interval).

8. Nadaljujemo s kodiranjem. Ko se najpomembnejši števki v $mLow$ in $mHigh$ ujemata, pošljemo ujemajočo se števko na izhod, za njo pa u ničel ali devetk.

Primer razširjanja delovnega intervala vidimo v razpredelnici 5.14.

Razpredelnica 5.14: Razširjanje delovnega intervala

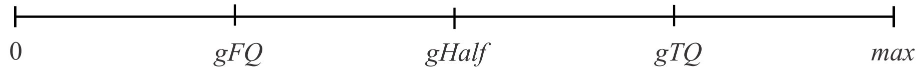
	nevarnost zaznana	nevarnost odpravljena
$mLow$	39810	38100
$mHigh$	40344	43449
u	0	1

5.4.2.2 Aritmetično kodiranje s skaliranjem

Kot smo že spoznali, je ključna težava aritmetičnega kodiranja ožanje delovnega intervala in posledično bližanje vrednosti $mLow$ in $mHigh$. Ko se vrednosti zaradi končne aritmetike digitalnega računalnika zlijeta v eno predstavljivo vrednost, nadaljnje kodiranje ni več mogoče. Aritmetično kodiranje s pomikanjem, ki smo ga spoznali v prejšnjem podpoglavju, težavo odpravi z odstranjevanjem najbolj pomembnih števk, ki se več ne spreminjajo, ter z dodatnim pravilom, ki najprej zazna, da sta vrednosti $mLow$ in $mHigh$ preblizu, in nato razmakne delovni interval.

Roko na srce, rešitev ni elegantna. Zato so razvili drugačen način aritmetičnega kodiranja, ki ne opravlja pomikanja, ampak skrbi samo za to, da je **delovni interval varno širok** [37] (videli bomo, da je delovni interval

dovolj širok takrat, ko je širši kot vsaj četrtina intervala $\mathcal{I}$). V vsakem koraku preverimo njegovo širino in ga, če je treba, razširimo oz. skaliramo (angl. scale). Skaliranje opravimo z različnimi funkcijami. Kodirnik zapiše, katero funkcijo smo v danem trenutku uporabili, to informacijo pa uporabi dekodirnik za vzdrževanje enakega delovnega intervala, ki potem omogoča dekodiranje. V ta namen interval $\mathcal{I}$ opremimo z dodatnimi stražarji (slika 5.15):



Slika 5.15: $gHalf$ označuje sredino intervala $\mathcal{I}$

- max: zgornja meja intervala; praviloma uporabimo kar največjo pozitivno vrednost celoštevilske spremenljivke;
- $gHalf$: stražar na polovici intervala; njegova vrednost je določena z enačbo 5.9;

$$gHalf = \left\lfloor \frac{max + 1}{2} \right\rfloor \tag{5.9}$$

- gFQ (angl. gFirstQuarter): stražar na četrtini intervala, njegovo vrednost izračunamo z enačbo 5.10 in

$$gFQ = \left\lfloor \frac{gHalf}{2} \right\rfloor \tag{5.10}$$

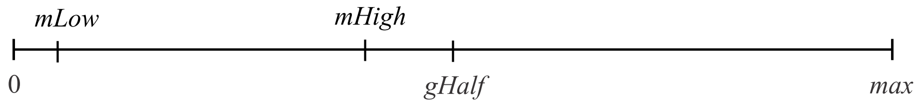
- gTQ (angl. gThirdQuarter): stražar na tretji četrtini intervala, postavljen z enačbo 5.11.

$$gTQ = 3\ gFQ \tag{5.11}$$

Kot že vemo, se v postopku aritmetičnega kodiranja delovni interval $[mLow,\ mHigh)$ oži. Da ne bi postal preozek, ukrepamo v naslednjih primerih:

1. $mHigh < gHalf$ (slika 5.16): intervala razširimo z enačbo 5.12, ki jo imenujemo **skaliranje E1**, ter označimo z bitom `0`, ki ga pošljemo na izhod.

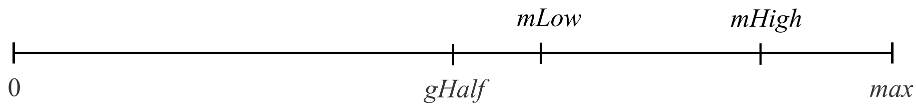
$$\begin{aligned} mLow &= 2\ mLow \\ mHigh &= 2\ mHigh + 1 \end{aligned} \tag{5.12}$$



Slika 5.16: Situacija, ko uporabimo skaliranje E1

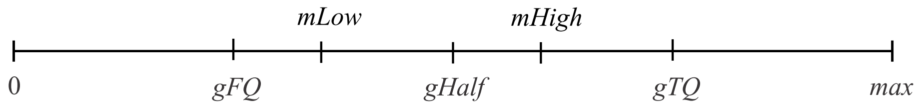
2. $mLow \geq gHalf$ (slika 5.17): uporabimo **skaliranje E2** (enačba 5.13), na izhod pa pošljemo bit 1.

$$\begin{aligned} mLow &= 2\,(mLow - gHalf) \\ mHigh &= 2\,(mHigh - gHalf) + 1 \end{aligned} \tag{5.13}$$



Slika 5.17: Situacija, ko uporabimo skaliranje E2

3. $mLow \geq gFQ \wedge mHigh < gTQ)$ (slika 5.18) uporabimo **skaliranje E3** (enačba 5.14).



Slika 5.18: Situacija, ko uporabimo skaliranje E3

$$\begin{aligned} mLow &= 2\,(mLow - gFQ) \\ mHigh &= 2\,(mHigh - gFQ) + 1 \end{aligned} \tag{5.14}$$

Pri skaliranju E3 bitov ne pošiljamo na izhod, temveč povečamo števec $counterE3$. Bite pošljemo šele, ko smo ponovno opravili eno od skaliranj E1 ali E2, in sicer:

- po skaliranju E1 pošljemo bit 0, ki mu sledi $counterE3$ bitov 1 ter
- po skaliranju E2 pošljemo bit 1, ki mu sledi $counterE3$ bitov 0.

$counterE3$ zatem postavimo na vrednost 0. Dokaz, da je opisan postopek pravilen, najdemo v [37].

Skaliranja lahko v postopku kodiranja enega simbola uporabimo večkrat. Ko zakodiramo vse znake $\sigma_i \in I$, moramo tudi pri aritmetičnem kodiranju s skaliranjem poslati na izhod dodatne podatke, da bi lahko sporočilo v celoti dekodirali. Velja:

- če je $mLow < gFQ$, pošljemo na izhod bita `01`, ki jima sledi $counterE3$ bitov `1`,
- sicer zapišemo bita `10`, ki jima sledi $counterE3$ bitov `0`.

Povzemimo:

Inicializacija.

imamo sporočilo $I = \langle \sigma_i \rangle$
določimo abecedo $\Sigma = \{\sigma_i\}$
določimo verjetnosti $p(\sigma_i)$
razvrstimo σ_i na interval [0,0, 1,0)
določimo število bitov r v registru
Izračunamo:
$max = 2^{r-1} - 1$
$mLow = 0;$ $mHigh = max$
$gHalf = \left\lfloor \frac{max+1}{2} \right\rfloor$ $gFQ = \left\lfloor \frac{gHalf}{2} \right\rfloor$ $gTQ = 3\, gFQ$
Postavimo $counterE3 = 0$

E1.
$mLow = 2\, mLow$
$mHigh = 2\, mHigh + 1$
izhod: `0`+ $counterE3$ bitov `1`
$counterE3 = 0$

E2.
$mLow = 2\,(mLow - gHalf)$
$mHigh = 2\,(mHigh - gHalf) + 1$
izhod: `1`+ $counterE3$ bitov `0`
$counterE3 = 0$

E3.
$mLow = 2\,(mLow - gFQ)$
$mHigh = 2\,(mHigh - gFQ) + 1$

$counterE3 = counterE3 + 1$

Finalizacija.

Če ie $mLow < gFQ$, pošljemo na izhod bita 01, ki jima sledi $counterE3$ bitov 1,
sicer pošljemo na izhod bita 10, ki jima sledi $counterE3$ bitov 0.

Poglejmo si primer 5.28. Dolžina registra naj bo $r = 8$ bitov. Razpored znakov abecede vidimo na sliki 5.19.

(a) $I = \langle \texttt{volos} \rangle$
(b) $\Sigma = \{\texttt{l o s v}\}$
(c) $P = [0{,}2 \quad 0{,}4 \quad 0{,}2 \quad 0{,}2]$

Primer 5.28: Primer aritmetičnega kodiranja s pomikanjem: (a) vhodno zaporedje, (b) njegova abeceda in (c) vektor verjetnosti.

Slika 5.19: Položaj simbolov glede na njihove verjetnosti

Inicializacija.
$max = 111\,1111_2 = 127_{(10)}$
$mLow = 000\ 0000_2 = 0_{(10)}$
$mHigh = 111\ 1111_2 = 127_{(10)}$
$gHalf = 100\ 0000_2 = 64_{(10)}$
$gFQ = 010\ 0000_2 = 32_{(10)}$
$gTQ = 110\ 0000_2 = 96_{(10)}$
$counterE3 = 0$

Kodiranje $\sigma_0 = \texttt{v}$. Za izračun $mLow$ in $mHigh$ uporabimo enačbo 5.7. Dobimo:

$mLow = 102$	$mHigh = 127$	
Skaliranje E2		izhod: 1
$mLow = 76$	$mHigh = 127$	
Skaliranje E2		izhod: 1
$mLow = 24$	$mHigh = 127$	

Kodiranje $\sigma_1 = \mathsf{o}$.

$mLow = 44$	$mHigh = 85$	
Skaliranje E3		$counterE3 = 1$
$mLow = 24$	$mHigh = 107$	

Kodiranje $\sigma_2 = \mathsf{l}$.

$mLow = 24$	$mHigh = 39$	
Skaliranje E1		izhod: `01`
$counterE3 = 0$		
$mLow = 48$	$mHigh = 79$	
Skaliranje E3		$counterE3 = 1$
$mLow = 32$	$mHigh = 95$	
Skaliranje E3		$counterE3 = 2$
$mLow = 0$	$mHigh = 127$	

Kodiranje $\sigma_3 = \mathsf{o}$.

$mLow = 25$	$mHigh = 75$	

Kodiranje $\sigma_4 = \mathsf{s}$.

$mLow = 55$	$mHigh = 64$	
Skaliranje E3		$counterE3 = 3$
$mLow = 46$	$mHigh = 65$	
Skaliranje E3		$counterE3 = 4$
$mLow = 28$	$mHigh = 67$	

Finalizacija. Ker je $mLow < gFQ$ in ker je $counterE = 4$, pošljemo na izhod bite `011111`. Zaporedje bitov kaže primer 5.29.

$$O = \langle \mathtt{1101011111} \rangle$$

Primer 5.29: Izhod aritmetičnega kodiranja s skaliranjem

Dekodiranje posnema kodiranje v obratnem postopku. Dekodirnik inicializiramo na enak način kot kodirnik, pri čemer podatke o abecedi Σ in verjetnostih simbolov σ_i dobimo iz glave datoteke, kot vhod I pa sprejmemo zaporedje bitov kodirnika. Potrebujemo še register R dolžine $r - 1$ bitov. Z enačbo 5.8 izračunamo spremenljivko v, z vrednostjo katere dekodiramo simbol σ_i, ki ga pošljemo na izhod. Z uporabo vrednosti $mLow(\sigma_i)$ in $mHigh(\sigma_i)$ izračunamo meje delovnega intervala, ki ga, če je preozek, razširimo s skaliranji E1, E2 ali E3. Po vsakem skaliranju odštejemo vred-

nost ustreznega stražarja od trenutne vrednosti v registru R, nato pa bite v R premaknemo za eno mesto v levo (angl. shift left, SHL). Na izpraznjeno mesto najmanj pomembnega bita vstavimo naslednji bit iz vhodnega zaporedja s funkcijo $nextBit()$. Bit preberemo tudi pri skaliranju E3, saj tudi skaliranje E3 pošilja bite na izhod, a to pri kodiranju stori z zamikom. Povzemimo:

Inicializacija.

abeceda Σ mora biti kodirniku znana
verjetnosti $p(\sigma_i)$ morajo biti kodirniku znane
razvrsti $\sigma_i \in \Sigma$ na interval [0,0, 1,0)
r velikost registra R v bitih
v register R preberemo prvih $r-1$ bitov iz vhodnega zaporedja I
$max = 2^{r-1} - 1$
$counterE3 = 0$
$mLow = 0;$ $\quad mHigh = max$
$gHalf = \left\lfloor \frac{max+1}{2} \right\rfloor$ $\quad gFQ = \left\lfloor \frac{gHalf}{2} \right\rfloor$ $\quad gTQ = \lfloor 3\, gFQ \rfloor$

Izračun vrednosti v in dekodiranje simbola.

v izračunamo z enačbo 5.8.
Dekodiramo σ_i.
Če je treba, razširimo interval s skaliranji E1, E2 ali E3.

E1. $mLow = 2\, mLow$ $\quad mHigh = 2\, mHigh + 1$
$R = 2\, R + nextBit() = SHL(R) + nextBit()$

E2. $mLow = 2(mLow - gHalf)$ $\quad mHigh = 2(mHigh - gHalf) + 1$
$R = 2(R - gHalf) + nextBit() = SHL(R - gHalf) + nextBit()$

E3. $mLow = 2(mLow - gFQ)$ $\quad mHigh = 2(mHigh - gFQ) + 1$
$R = 2(R - gFQ) + nextBit() = SHL(R - gFQ) + nextBit()$

Ko smo prebrali vse bite iz vhodnega zaporedja I, nadaljujemo z dekodiranjem. Na mesto bita LSB vedno vrivamo bit 0 ali bit 1. S transformacijami in dekodiranjem nadaljujemo, dokler na prvih dveh najpomembnejših

mestih nista bita 01 ali 10, drugi biti pa so 0 ali 1. Dekodiranje s tem zaključimo.

V nadaljevanju si oglejmo primer dekodiranja našega sporočila iz primera 5.28. Pri dekodiranju postane zaporedje bitov O iz primera 5.29 vhodno zaporedje I. Poglejmo:

Inicializacija.
$I = \langle 1101011111 \rangle$
$r = 8$
$R = 110\ 1011_2 = 107_{(10)}$
$max = 111\ 1111_2 = 127_{(10)}$
$mLow = 000\ 0000_2 = 0_{(10)}$
$mHigh = 111\ 1111_2 = 127_{(10)}$
$gHalf = 100\ 0000_2 = 64_{(10)}$
$gFQ = 010\ 0000_2 = 32_{(10)}$
$gTQ = 110\ 0000_2 = 96_{(10)}$
$counterE3 = 0$

1. korak

$v = 107/128 = 0{,}83593$	Dekodiramo $\sigma_0 = \mathsf{v}$
$mLow = 102$	$mHigh = 127$
Skaliranje E2	$R = 2(107 - 64) + 1 = 87_{(10)} = 101\ 0111_{(2)}$
$mLow = 76$	$mHigh = 127$
Skaliranje E2	$R = 2(87 - 64) + 1 = 47_{(10)} = 010\ 1111_{(2)}$
$mLow = 24$	$mHigh = 127$

2. korak

$v = 23/104 = 0{,}22115$	Dekodiramo $\sigma_1 = \mathsf{o}$
$mLow = 44$	$mHigh = 85$
Skaliranje E3	$R = 2(47 - 32) + 1 = 31_{(10)} = 011\ 1111_{(2)}$
$mLow = 24$	$mHigh = 107$

3. korak

$v = 7/84 = 0{,}083333$	Dekodiramo $\sigma_2 = \mathsf{l}$
$mLow = 24$	$mHigh = 39$
Skaliranje E1	$R = 2(31) + 0 = 62_{(10)} = 011\ 1110_{(2)}$
$mLow = 48$	$mHigh = 79$
Skaliranje E3	$R = 2(62 - 32) + 0 = 60_{(10)} = 011\ 1110_{(2)}$
$mLow = 32$	$mHigh = 95$

Skaliranje E3 $R = 2(60 - 32) + 0 = 56_{(10)} = 011\ 1000_{(2)}$
$mLow = 0$ $mHigh = 127$

4. korak
$v = 56/128 = 0{,}4375$ Dekodiramo $\sigma_3 =$ `o`
$mLow = 25$ $mHigh = 75$

5. korak
$v = 31/51 = 0{,}60784$ Dekodiramo $\sigma_3 =$ `s`
$mLow = 55$ $mHigh = 64$
Skaliranje E3 $R = 2(56 - 32) + 0 = 48_{(10)} = 011\ 0000_{(2)}$
$mLow = 46$ $mHigh = 65$
Skaliranje E3 $R = 2(48 - 32) + 0 = 32_{(10)} = 010\ 0000_{(2)}$
$mLow = 28$ $mHigh = 67$

Zaključek. Ker sta MSB `01` in ostali biti v R `0` in ker smo prebrali vse bite iz I, zaključimo.

5.5 Stiskanje podatkov s slovarjem

Do poznih sedemdesetih let prejšnjega stoletja sta bila Shannon-Fanojev in Huffmanov postopek edina uporabna algoritma za stiskanje podatkov. Kot smo že spoznali, temeljita na statistični analizi pogostosti pojavljanja znakov σ_i v sporočilu I. Lempel in Ziv [39] pa sta leta 1977 predstavila popolnoma drugačno idejo. Namesto da bi se osredotočila na posamezni znak $\sigma_i \in I$, sta zamenjala podzaporedje $\nu \subseteq I$ z oznako ω_i. Da bi to bilo možno, sta uporabila **slovar**.

Slovar $\mathcal{D} = \{(\zeta_i,\ \omega_i)\}$ hrani zaporedja ζ_i in oznake ω_i; zaporedja bomo imenovali **fraze**, oznake pa **žetoni**. Če za podzaporedje ν obstaja v slovarju fraza ζ_i in ima njej prirejen žeton ω_i manjše število bitov, kot je število bitov v ν, bomo dosegli stiskanje. Rodila se je nova paradigma stiskanja podatkov – **stiskanje podatkov s slovarjem** (angl. dictionary based compression), ki je nudila obilo možnosti za izboljšave na vseh treh komponentah: izgradnji in nadzoru slovarja, konstrukciji in iskanju fraz ter sestavi žetona. Posledično je bilo razvitih veliko metod. Odličen pregled najdemo v [30, 31, 32].

Slovarje, ključne entitete tega načina stiskanja, delimo na:

- statične (angl. static dictionaries) in

- dinamične (angl. dinamic dictionaries), tudi slovarje s prilagajanjem (angl. adaptive dictionaries).

Statične slovarje ustvarimo pred stiskanjem podatkov in se med stiskanjem ne spreminjajo. Tipična statična slovarja sta Slovar slovenskega knjižnega jezika ali šifrant avtomobilskih delov. Slabosti statičnih slovarjev za namene stiskanja podatkov so očitne:

- v slovarju je zelo veliko fraz, zato je žeton zelo velik; v procesu stiskanja podatkov bomo uporabili le delček njih, zato bo učinkovitost stiskanja majhna;
- statični slovar mora biti znan tako kodirniku kot dekodirniku; običajno je prevelik, da bi ga imelo smisel prenašati s stisnjenim sporočilom;
- posodabljanje statičnega slovarja zahteva, da ga morajo prejeti in posodobiti vsi uporabniki;
- za razširjanje starejših sporočil moramo hraniti tudi starejše različice slovarjev;
- slovarji so domenskoodvisni;
- podzaporedij ν, za katere ne najdemo ujemanja z nobeno frazo ζ, ne moremo stisniti oziroma moramo v takšnem primeru predvideti smiselni ubežni mehanizem;
- kombinacija posameznih fraz, ki bi morda povečala uspešnost stiskanja, ni možna.

Ker so statični slovarji zasnovani za druge namene, je bilo za stiskanje podatkov treba razviti drugačen koncept – koncept **dinamičnega slovarja** (v nadaljevanju bomo dinamičen slovarju imenovali slovar $\mathcal{D}$). $\mathcal{D}$ je pred začetkom stiskanja napolnjen samo z osnovnimi frazami ζ_i in njim prirejenimi žetoni ω_i ali pa je celo popolnoma prazen. Med procesom stiskanja algoritem gradi nove fraze glede na že videno sporočilo in s tem nove žetone. Posledično so ključne naloge, ki jih mora opraviti vsak algoritem stiskanja podatkov s slovarjem, naslednje:

- pomikanje skozi sporočilo $I = \langle \sigma_i \rangle$ in konstrukcija podzaporedja $\nu = \langle \sigma_i \rangle$, $\nu \subseteq I$, za katerega obstaja najdaljše ujemanje s frazo $\zeta_i \in \mathcal{D}$,
- zapis žetona ω_i, ki pripada frazi $\zeta_i = \nu$, na izhod O,

- konstrukcija nove fraze ζ_i in novega žetona ω_i in
- ažuriranje slovarja $\mathcal{D}$.

V nadaljevanju si bomo ogledali štiri najbolj znane algoritme stiskanja podatkov s slovarjem.

5.5.1 Algoritem LZ77

Lempel in Ziv sta leta 1977 (od tod tudi kratica LZ77) predlagala pristop, pri katerem sta del že videnega sporočila uporabila kot slovar [39]. Postopek sta realizirala z **drsečim oknom** (angl. sliding window), ki sta ga razdelila na dva dela (princip drsečega okna smo spoznali že v poglavju 2):

- prvi, večji del, je slovar $\mathcal{D}$,
- drugi, občutno manjši, je primerjalni pomnilnik in vsebuje del sporočila, ki ga bomo kodirali.

Poglejmo si primer 5.30. Primerjalni pomnilnik (njegova vsebina je prikazana z zelenimi črkami) hrani 10 znakov, drseče okno pa se je že pomaknilo čez prve štiri znake sporočila. Kodirnik poišče najdaljše ujemanje začetka zaporedja iz primerjalnega pomnilnika, zapisanega zeleno, z besedilom v slovarju (rdeče črke). Ujemanje najde na položaju 11 v dolžini 7 znakov, $\nu = \langle$ez tri $\rangle$. LZ77 zatem sestavi žeton $\omega = (11, 7, z)$. Žeton je torej definiran s trojčkom $\omega = (o,\ l,\ \sigma_\nu)$, kjer je:

- o odmik (angl. offset),
- l dolžina ujemanja (angl. length) in
- σ_ν znak iz primerjalnega pomnilnika, ki je povzročil neujemanje.

Ko smo na izhod zapisali žeton, premaknemo drseče okno za $l+1$ znakov in postopek ponavljamo, dokler ne pridemo do konca sporočila.

```
     0123456789012345678901234
⟨Čez tri gore, čez tri vode, čez tri zelene travnike⟩
                            0123456789
```

Primer 5.30: Slovar (rdeče črke) in vmesni pomnilnik (zelene črke) pri algoritmu LZ77 sestavljata znake v drsečem oknu

LZ77 ima kar nekaj slabosti, najpomembnejše pa so naslednje:

1. Slovar LZ77 se *pozablja.* Znakov, ki jih je drseče okno slovarja že prešlo, ne more več uporabiti. Seveda čas, v katerem je nastal LZ77, to opravičuje; računalniki so v sedemdesetih letih prejšnjega stoletja imeli zelo malo pomnilnika.

2. Iskanje najdaljše fraze je počasno, kar v praksi tudi danes omejuje dolžino drsečega okna, ki je od 4 k do 32 k zlogov.

3. Če je d dolžina slovarja, potem prvih d simbolov ne moremo kodirati, ampak jih neposredno zapišemo na izhod.

4. Žeton je sestavljen zelo negospodarno. Če je slovar dolžine 4 k zlogov, dolžina primerjalnega pomnilnika pa 256 zlogov, žeton sestoji iz 12 bitov za zapis odmika, 8 bitov za zapis dolžine ujemanja in 8 bitov za kodiranje naslednjega znaka po shemi ASCII, skupaj torej kar 28 bitov, kar je še posebej neugodno, ko ne najdejo ujemanja (takrat je žeton $\omega = (0, 0, z)$).

Razširjanje je pri algoritmu LZ77 zelo preprosto. Najprej preberemo prvih d znakov in z njimi napolnimo slovar, nato pa beremo žetone. Glede na njihovo vsebino kopiramo znake iz $\mathcal{D}$ na izhod. Zatem premaknemo drseče okno za $l + 1$ znakov, preberemo nov žeton in postopek ponovimo.

Obstaja množica izpeljank algoritma LZ77, na primer LZSS, LZB, SLH, LZARI, LZX, LZRW1, LZRW4. Bralec najde razlage njihovega delovanja v [30]. Med njimi je tudi Deflate, eden najbolj popularnih algoritmov za stiskanje podatkov. Razvil ga je P. W. Katz [40] in uporabil v dobro znanem programu PkZip, ki ga prodaja s pomočjo podjetja PKWARE, Inc. Gailly in Adler sta Deflate nato implementirala v knjižnicah ZLIB in GZIP. Knjižnici nista pod nobenim patentom in ne zahtevata nobenega licenciranja. Deflate je del protokola HTTP ter formatov PDF, PNG in MNG [30]. V nadaljevanju si bomo ogledali izpeljanko LZ77, to je LZSS.

5.5.2 Algoritem LZSS

Storer in Szymanski sta predlagala izboljšano metodo LZ77, znano kot LZSS. Da sta se izognila dragemu žetonu $(0, 0, \sigma_\nu)\}$, sta ukrepala na naslednji način:

- Če nismo našli dovolj dolgega ujemanja l, $l < t$, kjer je t uporabniško podan parameter, pošljemo na izhod prvi znak iz iskalnega pomnilnika σ_ν, kodiran s kodo dane abecede (običajno kot znak ASCII), ki mu kot predpono dodamo kontrolni bit 0. Žeton je v tem primeru $\omega_i = (\mathtt{0}, \sigma_\nu)$.

- Če smo našli dovolj dolgo ujemanje, $l \geq t$, je $\omega_i = (\mathtt{1}, o, l)$.

(a) $I = \langle$`rabarbararabarbara`$\rangle$
(b) $\Sigma = \{$`a b r`$\}$

Primer 5.31: (a) Vhodno zaporedje, (b) abeceda

Oglejmo si primer 5.31, pri čemer naj bo velikost slovarja 8 znakov, velikost primerjalnega pomnilnika 4 znake, najkrajša dolžina ujemanja, ko pošljemo na izhod daljši žeton $\omega_i = (1, o, l)$, pa naj bo $t = 2$ znaka. Postopek kodiranja prikažemo s primerom 5.32. Najprej napolnimo primerjalni pomnilnik s prvimi štirimi znaki sporočila I. V slovarju $\mathcal{D}$ še ni nobenega znaka, zato ujemanja ne najdemo. Na izhod pošljemo kontrolni bit 0 in znak r (prva vrstica v primeru 5.32). Znake v iskalnem pomnilniku premaknemo za število kodiranih znakov v levo (v našem primeru za eno mesto), prav toliko novih znakov pa vstavimo v iskalni pomnilnik z desne (druga vrstica v primeru 5.32). V vrstici 4 prvič najdemo ujemanje, in sicer prvi znak v iskalnem pomnilniku a se ujema z znakom na položaju 1 v slovarju. A ker je dolžina ujemanja manjša kot t, tudi tokrat pošljemo na izhod krajši žeton. Daljše ujemanje najdemo v vrstici 6. Dolžina ujemanja je $l = 3$, začne pa se na položaju 2. Tvorimo ustrezen žeton in opravimo pomik za tri znake, torej za l mest. Ko se $\mathcal{D}$ napolni, skrajno levi simboli zdrsnejo iz slovarja.

```
          7 6 5 4 3 2 1 0 | 0 1 2 3     izhod
 1.                       | r a b a     (0,r)
 2.                     r | a b a r     (0,a)
 3.                   r a | b a r b     (0,b)
 4.                 r a b | a r b a     (0,a)
 5.               r a b a | r b a r     (0,r)
 6.             r a b a r | b a r a     (1,2,3)
 7.       r a b a r b a r | a r a b     (1,1,2)
 8.       b a r b a r a r | a b a r     (0,a)
 9.       a r b a r a r a | b a r b     (1,5,3)
10.       a r a r a b a r | b a r a     (1,2,3)
11.       r a b a r b a r | a           (0,a)
12.       a b a r b a r a |
```

Primer 5.32: Kodiranje z LZSS

LZSS v nasprotju z LZ77 vedno zapiše žetone (LZ77 je prvih d simbolov zapisal neposredno na izhod). Da bo primer verodostojnejši, si omislimo naslednje kodiranje žetonov:

- Ko je kontrolni bit 0, znake abecede Σ zakodiramo s kodo VLC, in sicer a $= 0$, b $= 10$ in r $= 11$.
- Ko je kontrolni bit 1, preostali del žeton $\omega_i = \{o, l\}$ zakodiramo s petimi biti; prvi trije bodo ustrezali odmiku o, zadnja dva pa dolžini ujemanja l, in sicer $l = 2$ kodirano z bitom 0, $l = 3$ z bitoma 10 in $l = 4$ z bitoma 11.

$$O = \langle 0110001000011101010100100011011010101000 \rangle$$

Primer 5.33: Kodiranje žetonov LZSS

	koda žetona	izhod	7	6	5	4	3	2	1	0
1.	011	⟨r⟩								r
2.	00	⟨a⟩							r	a
3.	010	⟨b⟩						r	a	b
4.	00	⟨a⟩					r	a	b	a
5.	011	⟨r⟩				r	a	b	a	r
6.	101010	⟨bar⟩	r	a	b	a	r	b	a	r
7.	10010	⟨ar⟩	b	a	r	b	a	r	a	r
8.	00	⟨a⟩	a	r	b	a	r	a	r	a
9.	110110	⟨bar⟩	a	r	a	r	a	b	a	r
10.	101010	⟨bar⟩	r	a	b	a	r	b	a	r
11.	00	⟨a⟩	a	b	a	r	b	a	r	a

Primer 5.34: Dekodiranje z LZSS

Izid kodiranja žetonov kaže primer (5.33). Postopek dekodiranja ponazorimo s primerom 5.34. Preberemo prvi bit. Ker je ta 0, vemo, da bo sledila koda znaka σ_i. Preberemo bit 1, zato vemo, da moramo prebrati še en bit in bita 11 dekodiramo kot r. r pošljemo na izhod in v slovar (prva vrstica v primeru 5.34). Na podoben način postopamo tudi v primeru vrstic 2 in 3, le da tokrat že opravljamo pomik znakov v slovarju. V vrstici 6 je kontrolni bit 1, zato preberemo preostali žeton $\omega_i = (o, l)$ v dolžini 5 bitov. Prvi trije biti določajo odmik $o = 2$, nato dekodiramo še dolžino ujemanja $l = 3$. Pogledamo v slovar. Aktualen slovar je v vrstici 5. Na položaju 2 najdemo frazo v dolžini treh znakov ⟨bar⟩. Izpišemo jo na izhod in z desne vstavimo v slovar. Na ta način nadaljujemo z dekodiranjem, dokler nismo prebrali vseh žetonov.

Storer in Szymanski sta poskrbela tudi za hitrejše iskanje najdaljšega ujemanja. Slovar $\mathcal{D}$ sta v ta namen predstavila z večvejnim drevesom.

5.5.3 Algoritem LZ78

Lempel in Ziv sta eno leto po objavi algoritma LZ77 predlagala drugačno tehniko stiskanja s slovarjem [41]. Odmaknila sta se od paradigme drsečega okna in predpostavila, da količina pomnilnika ni več ovira za zasnovo drugačnega koncepta, imenovanega LZ78.

Pri algoritmu LZ78 kodirnik in dekodirnik začneta stiskati s praznim slovarjem. Vsak prebran znak sporočila $\sigma_i \in I$ dodamo trenutnemu zaporedju $\nu = \nu + \sigma_i$ (na začetku je $\nu = \langle\rangle$). Proces nadaljujemo, dokler se ν ujema s frazo $\zeta_i \in \mathcal{D}$. Ko ujemanja več ne najdemo, pošljemo na izhod žeton $\omega_i = (i, \sigma_\nu)$, kjer je i indeks fraze ζ_i, $\sigma_\nu \in I$ pa znak, ki je povzročil neujemanje. Nato sestavimo novo frazo $\zeta_j = \nu + \sigma_\nu$, ji priredimo še neuporabljen indeks j in oba vstavimo v $\mathcal{D}$. Indeks 0 je rezerviran za primere, ko je dolžina zaporedja $|\nu| = 1$, in predstavlja ubežno kodo.

Delovanje algoritma si najlažje ogledamo s primerom 5.31, postopek kodiranja pa demonstrira primer 5.35. V vrstici 1 inicializiramo slovar, nato

	kodiranje		slovar	
	ν	ω	i	ζ
1			0	⟨ESC⟩
2	⟨r⟩	(0, r)	1	⟨r⟩
3	⟨a⟩	(0, a)	2	⟨a⟩
4	⟨b⟩	(0, b)	3	⟨b⟩
5	⟨ar⟩	(2, r)	4	⟨ar⟩
6	⟨ba⟩	(3, a)	5	⟨ba⟩
7	⟨ra⟩	(1, a)	6	⟨ra⟩
8	⟨rab⟩	(6, b)	7	⟨rab⟩
9	⟨arb⟩	(4, b)	8	⟨arb⟩
10	⟨ara⟩	(4, a)	9	⟨ara⟩

Primer 5.35: Kodiranje z LZ78

začnemo kodirati. Ker fraze ⟨r⟩ še ni v $\mathcal{D}$, pošljemo na izhod O žeton $\omega = (0\mathsf{a})$. Indeks 0 ustreza ubežni kodi ESC, ki nam pove, da znaka še ni v slovarju in da bo sledila koda ASCII. ⟨r⟩ postane prva fraza v $\mathcal{D}$, ki ji priredimo indeks 1 (glej vrstico 2). Na enak način zakodiramo tudi naslednja znaka sporočila, to sta a in b. Nato se med branjem I znova spoprimemo z znakom a. Ker ⟨a⟩ kot fraza v slovarju že obstaja, preverimo naslednji znak. Trenutno zaporedje $\nu =$⟨ar⟩ v slovarju še ne obstaja, zato pošljemo na izhod kodo najdaljše fraze, ki se je ujemala z ν, to je koda fraze $\zeta =$⟨a⟩, ki je 2. Žeton, ki ga zapišemo na izhod O, je torej 2r. Novo frazo ⟨ar⟩ zapišemo

v slovar in ji priredimo naslednji prosti indeks. Postopek nadaljujemo, dokler ne pridemo do konca sporočila. O hrani zaporedje žetonov, ki ga kaže primer 5.36.

$$O = \langle \texttt{(0,r)(0,a)(0,b)(2,r)(3,a)(1,a)(6,b)(4,b)(4,a)} \rangle$$

Primer 5.36: Rezultat kodiranja LZ78

Postopek dekodiranja je popolnoma identičen in ga kaže primer 5.37. Poznati moramo samo zaporedje žetonov O ter pripraviti slovar, v katerem je samo en simbol. To je ubežna koda, ki ji je prirejen indeks 0 (vrstica 1). Preberemo prvi žeton (0, r). Ker je indeks v žetonu 0, vemo, da sledi znak ASCII, ki ga zapišemo na izhod in vstavimo v slovar; priredimo mu indeks 1. Na enak način dekodiramo tudi znaka a in b. Naslednji žeton je $\omega = (2, \mathsf{r})$. Iz slovarja dobimo frazo, ki je prirejena indeksu 2, to je $\zeta = \langle \mathsf{a} \rangle$, ki mu prilepimo simbol iz žetona, to je r. Dekodiramo $\langle \mathsf{ar} \rangle$, ki ga zapišemo na izhod in vstavimo v slovar.

	dekodiranje		slovar	
	ω	izhod	i	ζ
1			0	⟨ESC⟩
2	(0, r)	⟨r⟩	1	⟨r⟩
3	(0, a)	⟨a⟩	2	⟨a⟩
4	(0, b)	⟨b⟩	3	⟨b⟩
5	(2, r)	⟨ar⟩	4	⟨ar⟩
6	(3, a)	⟨ba⟩	5	⟨ba⟩
7	(1, a)	⟨ra⟩	6	⟨ra⟩
8	(6, b)	⟨rab⟩	7	⟨rab⟩
9	(4, b)	⟨arb⟩	8	⟨arb⟩
10	(4, a)	⟨ara⟩	9	⟨ara⟩

Primer 5.37: Dekodiranje z LZ78

Implementacije LZ78 se razlikujejo glede na slovar, pri čemer moramo biti pozorni na:

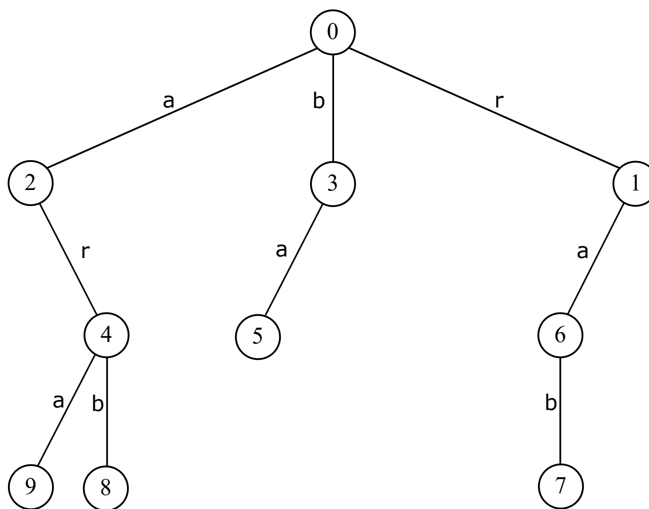
- število bitov, ki jih bomo namenili žetonu;
- hitrost iskanja fraze.

Ker kodirnik in dekodirnik tvorita slovar popolnoma usklajeno, lahko število bitov prilagajamo glede na število fraz, ki jih vsebuje slovar. Teoretično naj bi LZ78 stiskal podatke bolje, čim več fraz je v slovarju. Izkaže

pa se, da to ne velja. Večje število fraz zahteva več bitov za indeks fraze, hkrati pa v slovarju obstaja veliko fraz, ki se zelo redko uporabijo (ali celo nikoli več). Zato spremljamo uspešnost stiskanja. Ko ugotovimo, da se ta začne zmanjševati, ukrepamo z eno od dveh strategij:

- slovar v celoti izpraznimo in ga začnemo graditi znova ali
- iz slovarja odstranimo najmanj pogosto uporabljene fraze; temu postopku pravimo klestenje fraz.

Slovar lahko implementiramo na različne načine. Najpogosteje uporabljamo sekljalne tabele ali iskalna drevesa (na primer številska drevesa, ki jih bomo spoznali pozneje). Številsko drevo za slovar $\mathcal{D}$ iz primera 5.37 vidimo na sliki 5.20). Vsako vozlišče ima toliko možnih potomcev, kot je moč abecede $|\Sigma|$. V vsakem vozlišču hranimo indeks fraze, ki jo dobimo z lepljenjem simbolov na vejah drevesa, ko se pomikamo od korena drevesa.



Slika 5.20: Številsko drevo kot slovar pri LZ78

5.5.4 Algoritem LZW

Algoritem LZ78 je doživel množico izboljšav in komercialnih uporab. Najbolj znana nadgradnja je algoritem LZW, ki ga je leta 1984 predlagal Terry Welch in ga tudi patentiral [42]. Algoritem LZW najdemo pri formatih GIF in TIFF ter pri starejših datotekah PDF.

Pred začetkom kodiranja slovar $\mathcal{D}$ inicializiramo z vsemi znaki $\sigma_i \in \Sigma$ (pogosto kar s celotnim naborom/abecedo ASCII). Posledično ne potrebujemo ubežnega mehanizma, saj na izhod nikoli ne pošljemo nekodiranega znaka. Tudi žeton je zato preprost in sestoji samo iz indeksa fraze. V procesu kodiranja iz vhodnega zaporedja I preberemo naslednji znak σ_i in ga dodajamo zaporedju ν, ki mu pri LZW pravimo *trenutna beseda*. Ko preberemo σ_i, ki bi, če bi ga pripisali k ν, tvoril novo, še neznano frazo, pošljemo žeton, ki ustreza trenutni vrednosti v ν, v slovar pa vstavimo novo frazo, sestavljeno kot $\nu + \sigma_i$. Trenutno besedo nato postavimo na $\nu = \sigma_i$, torej na znak, ki je povzročil neujemanje s frazami v $\mathcal{D}$.

Postopek kodiranja bomo najlažje pokazali s primerom 5.31, korake pa kaže primer 5.39. V korakih 1, 2 in 3 vstavimo znake $\sigma_i \in \Sigma$ v $\mathcal{D}$, inicializiramo trenutno besedo $\nu = \langle\rangle$, nato pa začnemo kodirati. Preberemo prvi znak $\sigma_0 = \mathsf{r}$ in preverimo, ali $\nu + \sigma_0$ obstaja v $\mathcal{D}$. Ker je odgovor pritrdilen, postane $\nu = \nu + \sigma_0 = \langle\mathsf{r}\rangle$. Nato preberemo naslednji znak $\sigma_1 = \mathsf{a}$. Tudi tokrat preverimo, ali fraza $\nu + \sigma_1 = \langle\mathsf{ra}\rangle$ v slovarju obstaja. Ker ne obstaja, storimo naslednje:

- na izhod O pošljemo žeton ω fraze $\nu = \zeta$ ($\omega = (1)$ v konkretnem primeru);
- tvorimo novo frazo $\zeta = \nu + \sigma_i = \langle\mathsf{ra}\rangle$ in jo vstavimo v slovar ter ji dodelimo še neuporabljen žeton $\omega = (4)$;
- trenutno besedo ν postavimo na vrednost σ_i ($\nu = \langle\mathsf{a}\rangle$) in
- začnemo z naslednjim korakom algoritma.

V koraku 5 je $\nu = \langle\mathsf{a}\rangle$, naslednji znak iz I pa je $\sigma_2 = \mathsf{b}$. Fraza $\zeta = \nu$ obstaja v $\mathcal{D}$, fraza $\nu + \sigma_2$ pa ne. Zato pošljemo na izhod žeton zadnje najdene fraze, novo frazo $\langle\mathsf{ab}\rangle$ vstavimo v $\mathcal{D}$, postavimo $\nu = \langle\sigma_2\rangle$ ter preberemo naslednji znak iz I. Algoritem tako nadaljuje do koraka 9, v katerega vstopi z vrednostjo trenutne besede $\nu = \langle\mathsf{b}\rangle$. Naslednji znak $\sigma_6 = \mathsf{a}$. Ker fraza $\langle\mathsf{ba}\rangle$ že obstaja v $\mathcal{D}$, postane tudi $\nu = \langle\mathsf{ba}\rangle$, algoritem pa prebere nov znak iz I, $\sigma_7 = \mathsf{r}$. Trenutna beseda, razširjena s simbolom b, je $\langle\mathsf{bar}\rangle$, ta fraza pa v $\mathcal{D}$ še ne obstaja. Zato na izhod pošljemo žeton fraze $\zeta = \langle\mathsf{ba}\rangle$, vstavimo novo frazo $\nu + \sigma_7 = \langle\mathsf{bar}\rangle$ v $\mathcal{D}$ ter inicializiramo trenutno besedo ν s simbolom, ki je povzročil neujemanje fraze, torej $\nu = \sigma_7 = \langle\mathsf{r}\rangle$. Ko preberemo vse znake

$$O = \langle \mathtt{(3)(1)(2)(1)(3)(6)(4)(4)(9)(9)(1)} \rangle$$

Primer 5.38: Žetoni LZW

iz I, dobimo na izhodu zaporedje žetonov, ki jih kaže primer 5.38.

	kodiranje			slovar	
	ν	σ_i	ω	i	ζ
1				1	⟨a⟩
2				2	⟨b⟩
3				3	⟨r⟩
4	⟨r⟩	a	(3)	4	⟨ra⟩
5	⟨a⟩	b	(1)	5	⟨ab⟩
6	⟨b⟩	a	(2)	6	⟨ba⟩
7	⟨a⟩	r	(1)	7	⟨ar⟩
8	⟨r⟩	b	(3)	8	⟨rb⟩
9	⟨ba⟩	r	(6)	9	⟨bar⟩
10	⟨ra⟩	r	(4)	10	⟨rar⟩
11	⟨ra⟩	b	(4)	11	⟨rab⟩
12	⟨bar⟩	b	(9)	12	⟨barb⟩
13	⟨bar⟩	a	(9)	13	⟨bara⟩
14	⟨a⟩	/	(1)	/	/

Primer 5.39: Kodiranje z LZW

Kot smo videli, kodirnik sestavi novo frazo z naslednjim znakom iz I; pravimo, da kodirnik s sestavljanjem fraz *prehiteva*. To prehitevanje mora dekodirnik upoštevati, zato bo s tvorjenjem novih fraz za en znak *zaostajal*. Postopek dekodiranja predstavimo s primerom 5.40. V inicializaciji najprej napolnimo slovar z znaki abecede $\sigma_i \in \Sigma$, nato pa v koraku 4 preberemo prvi žeton $\omega =$(3) iz vhoda I. Slovar vrne frazo $\zeta =$⟨r⟩, ki jo pošljemo na izhod O. V tem koraku dekodiranja nove fraze, ki bi jo vstavili v $\mathcal{D}$, še ne moremo sestaviti, si pa dekodirano frazo shranimo v trenutno besedo ν. V koraku 5 preberemo žeton $\omega =$(1), ki ga s pomočjo $\mathcal{D}$ dekodiramo kot frazo ⟨a⟩. Tako lahko sestavimo novo frazo s pomočjo trenutne besede $\nu =$⟨r⟩ in prve črke dekodirane fraze, to je $\zeta = \nu + \sigma$ (rdeče črke v vrsticah 4 in 5). Novo frazo vstavimo v slovar, trenutna beseda ν pa dobi vrednost dekodirane fraze. Nato postopek ponovimo z naslednjim žetonom. Oglejmo si še korak v vrstici 10. Žeton (4) ustreza frazi $\zeta =$ ⟨ra⟩, ki jo najprej zapišemo na izhod, nato pa tvorimo novo frazo s trenutno besedo $\nu =$ ⟨ba⟩ in prvo črko dekodirane fraze r. Novo frazo $\zeta =$ ⟨bar⟩ vstavimo v slovar, dekodirana fraza pa postane nova trenutna beseda ν.

	dekodiranje			slovar	
	ω	O	ν	i	ζ
1				(1)	⟨a⟩
2				(2)	⟨b⟩
3				(3)	⟨r⟩
4	(3)	⟨r⟩	⟨r⟩		
5	(1)	⟨a⟩	⟨a⟩	(4)	⟨ra⟩
6	(2)	⟨b⟩	⟨b⟩	(5)	⟨ab⟩
7	(1)	⟨a⟩	⟨a⟩	(⟨6⟩)	⟨ba⟩
8	(3)	⟨r⟩	⟨r⟩	(⟨7⟩)	⟨ar⟩
9	(6)	⟨ba⟩	⟨ba⟩	(⟨8⟩)	⟨rb⟩
10	(4)	⟨ra⟩	⟨ra⟩	(⟨9⟩)	⟨bar⟩
11	(4)	⟨ra⟩	⟨ra⟩	(10)	⟨rar⟩
12	(9)	⟨bar⟩	⟨bar⟩	(11)	⟨rab⟩
13	(9)	⟨bar⟩	⟨bar⟩	(12)	⟨barb⟩
14	(1)	⟨a⟩	⟨a⟩	(13)	⟨bara⟩

Primer 5.40: Dekodiranje z LZW

5.6 Kodiranje zaporedij celih števil

Cela števila so poleg znakov najpogostejši osnovni podatkovni tip, ki jih je treba stiskati. V tem podpoglavju si bomo najprej pogledali najpogosteje uporabljeno tehniko – Golombovo kodiranje in njeno izpeljanko Golomb-Riceovo kodiranje, sledilo pa bo nekoliko manj znano prilagodljivo binarno zaporedno kodiranje. Interpolativno kodiranje, ki si ga bomo ogledali ob koncu podpoglavja, pa je že tehnika, ki je v nekaterih primerih celo boljša od aritmetičnega kodiranja.

5.6.1 Golombovo kodiranje

Golomb je leta 1966 predstavil kodiranje števila ponovitev pri algoritmu stiskanja RLE [43], ki je, v splošnem, nekorelirano zaporedje nenegativnih celih števil. Izkazalo se je, da je njegov postopek optimalen v primeru geometrijske porazdelitve števil, ko je enako učinkovit kot Huffmanovo kodiranje, le da je hitrejši. Golombovo kodiranje potrebuje parameter k, s katerim najbolje prilegamo strmino funkcije geometrijske porazdelitve konkretnim podatkom. Primeren k najpogosteje določimo eksperimentalno, včasih pa posežemo tudi po hevristiki.

Golombova koda nenegativnega celega števila σ sestoji iz dveh delov: iz

količnika/kvocienta q in ostanka r (enačba 5.15).

$$\begin{aligned} q = & \left\lfloor \frac{\sigma}{k} \right\rfloor \\ r = & \sigma - q\,k \end{aligned} \tag{5.15}$$

Količnik q zakodiramo z unarno kodo, r pa s prisekano/modificirano binarno kodo. **Unarna/vejična koda** števila q sestoji iz q bitov 1, ki jim sledi vejica, to je bit 0 (vlogo 0 in 1 lahko zamenjamo). Če je $q = 3$, q z unarno kodo zapišemo kot $\langle 1110 \rangle$, če je $q = 9$, pa je unarni zapis $\langle 1111111110 \rangle$. Unarna predstavitev $q = 0$ je $\langle 0 \rangle$, torej samo vejica. Kot vidimo, je unarna koda smiselna le, če je q majhen.

Prisekana/modificirana binarna koda (angl. truncated binary code) lahko zapiše ostanek r nekoliko učinkoviteje, kot če bi uporabili klasičen binarni zapis. Naj bo abeceda Σ z $n = |\Sigma|$ simboli. Za zapis vsakega od n simbolov potrebujemo u bitov (enačba 5.16), če jih želimo zapisati v binarni obliki.

$$u = \lceil \log_2 n \rceil \tag{5.16}$$

Ravno zaokrožitev navzgor v enačbi 5.16 daje možnost sestave učinkovitejše kode. Modificirana binarna koda tako prvih $z = 2^u - n$ simbolov zapiše z $u - 1$ biti, preostalih $n - z$ simbolov pa z u biti. Oglejmo si primer 5.41, kjer je $n = 5$. Iz enačbe 5.16 dobimo $u = 3$. Modificirana binarna koda

$$\Sigma = \{0,\ 1,\ 2,\ 3,\ 4\}$$

Primer 5.41: Simboli abecede, ki jim bomo dodelili Golombove kode

prve $z = 2^u - n = 3$ simbole zapiše z $u - 1$ biti, torej: $\langle 00 \rangle$, $\langle 01 \rangle$, $\langle 10 \rangle$, preostala $n - z = 2$ simbola pa z $u = 3$ biti, in sicer v našem primeru z $\langle 111 \rangle$ in $\langle 110 \rangle$. Postopek konstrukcije prisekane binarne kode za ta primer vidimo v razpredelnici 5.15, za primer, ko je $n = 10$, pa so prisekane binarne kode podane v razpredelnici 5.16.

Vrnimo se h Golombovi kodi in sestavimo Golombovo kodo za število $\sigma = 23$, če je Golombov parameter $k = 5$. Hitro dobimo $q = 4$ in $r = 3$. Zaloga vrednosti ostanka je naša abeceda iz primera 5.41. Prisekane binarne kode za posamezne vrednosti dobimo iz razpredelnice 5.15. Golombova koda za število 23 pri Golombovem parametru $k = 5$ je torej $\langle 11110\!:\!110 \rangle$, kjer Golombovo kodo, kot je v literaturi običaj, zaradi preglednosti zapišemo v obliki $q\!:\!r$, čeprav : seveda ni del kode.

Oglejmo si še nekoliko drugačen primer. Naj bo zaporedje $I = \langle \sigma_i \rangle$, kjer $\sigma_i \in \Sigma$ (glej primer 5.42a in b). Predpostavimo, da želimo zakodirati

Razpredelnica 5.15: Konstrukcija prisekane binarne kode za $n = 5$

σ_i	binarna koda	prisekana binarna koda	komentar
0	000	00	izbrišemo MSB
1	001	01	izbrišemo MSB
2	010	10	izbrišemo MSB
/	011		ne uporabimo
/	100		ne uporabimo
/	101		ne uporabimo
3	110	110	
4	111	111	

Razpredelnica 5.16: Prisekana binarna koda za $n = 10$

σ_i	prisekana binarna koda
0	000
1	001
2	010
3	011
4	100
5	101
6	1100
7	1101
8	1110
9	1111

I z Golombovim kodiranjem, ko je Golombov parameter $k = 2$. Simbolom σ_i najprej priredimo nenegativne vrednosti $\sigma_i \in 0, 1, \cdots, |\Sigma| - 1$, pri čemer dobijo simboli, ki so pogostejši, manjšo vrednost. V našem primeru priredimo simbolu a vrednost 0, b vrednost 1, c 2, d 3 in e 4. Dobimo zaporedje nenegativnih števil J (glej primer 5.42c). Številkam določimo Golombove kode na naslednji način:

- Koda za 0: $q = 0$, $r = 0$; vejična koda je v tem primeru predstavljena kot $\langle 0 \rangle$, prav tako prisekana binarna koda; Golombova koda za a je torej $\langle 0{:}0 \rangle$.

(a) I=⟨aacbaeaaababcada⟩
(b) Σ ={a b c d e}
(c) J=$\langle \sigma_i \rangle = \langle$0 0 2 1 0 4 0 0 0 1 0 1 2 0 3 0$\rangle$

Primer 5.42: (a) Vhodno zaporedje, (b) abeceda, (c) zaporedje nenegativnih celih števil.

- Koda za 1: $q = 0$, $r = 1$, Golombova koda ⟨0:1⟩.
- Koda za 2: $q = 1$, $r = 0$, Golombova koda ⟨10:0⟩.
- Koda za 3: $q = 1$, $r = 1$, Golombova koda ⟨10:1⟩.
- Koda za 4: $q = 2$, $r = 0$, Golombova koda ⟨110:0⟩.

Zaporedje Golombovih kod pri $k = 2$ kaže primer 5.43.

$$O = \langle \texttt{0000100010011000000000100011000010100} \rangle$$

Primer 5.43: Bitno zaporedje, tvorjeno z Golombovim kodiranjem

Golombovo kodiranje lahko uporabimo tudi za kodiranje negativnih števil, ki pa jih moramo preslikati v pozitivna. Običajno negativna in pozitivna števila **prepletemo** tako, da negativna števila preslikamo v lihe, pozitivna pa v sode vrednosti. To dosežemo s preprosto transformacijo (enačba 5.17), kjer je σ_i vrednost, ki jo kodiramo, vrednost σ_i^+ pa zapišemo z Golombovim kodiranjem.

$$\sigma_i^+ = \begin{cases} 2\sigma_i, & \sigma_i \geq 0 \\ 2|\sigma_i| - 1, & \sigma_i < 0 \end{cases} \tag{5.17}$$

5.6.2 Golomb-Riceovo kodiranje

Posebni primer Golombovega kodiranja je Golomb-Riceovo kodiranje [44] (krajše Riceovo kodiranje). V primeru Riceovega kodiranja je parameter k vedno potenca števila 2. Določanje Riceove kode opravimo na naslednji način:

- najprej določimo predznak in zanj namenimo en bit, ki postane MSB Riceove kode;
- odstranimo k najmanj pomembnih bitov (LSBs), ki postanejo najmanj pomembni biti Riceove kode;

- preostale bite, poimenovali jih bomo ostanek r, zapišemo z vejično kodo in so srednji del Riceove kode (vejica pri Riceovi kodi je najpogosteje predstavljena z bitom 1).

Oglejmo si nekaj primerov določanja Riceove kode, ko je $k = 2$:

- število $\sigma = 0_{10}$, 0_2: predznak = $\langle 0 \rangle$, LSBs = $\langle 00 \rangle$, $r = \langle 0 \rangle$. Riceova koda je $\langle 0:1:00 \rangle$. Razložimo: prva ničla je predznak, ko smo odstranili k bitov, od vhodne vrednosti ni ostalo nič bitov za ostanek, torej vejična koda sestoji samo iz vejice, to je bita 1, na koncu pripišemo še k bitov 0.

- število $\sigma = -7_{10}$, 111_2; predznak = $\langle 1 \rangle$, LSBs = $\langle 11 \rangle$, $r = \langle 1 \rangle$; Riceova koda $\langle 1:01:11 \rangle$.

- število $\sigma = 13_{10}$, 1101_2; predznak = $\langle 0 \rangle$, LSBs = $\langle 01 \rangle$, $r = 3$; Riceova koda $\langle 0:0001:01 \rangle$.

Riceove kode določimo le z nekaj binarnimi operacijami, zato je Riceovo kodiranje zelo hitro. Zelo pogosto je pri kodirnikih zvoka (FLAC [30], SHORTEN [45], MPEG-4 ALS[46]).

5.6.3 Prilagodljivo binarno zaporedno kodiranje

Metodo prilagodljivega binarnega zaporednega kodiranja (angl. Binary Adaptive Sequential Coding, BASC) sta razvila Moffat in Anh [47]. Namenjena je kodiranju zaporedja nenegativnih celih števil $I = \langle \sigma_i \rangle$. Metoda je zelo preprosta in uporablja dve delovni spremenljivki:

- b_p hrani število bitov, ki smo jih potrebovali za zapis predhodnega števila;

- b_a pa število bitov, ki bi jih potrebovali, da bi zapisali trenutno število.

Začetno število bitov b_p poda uporabnik, lahko pa ga določimo tudi s kakšno hevristiko (lahko poiščemo povprečno vrednost števil $\overline{\sigma}$, ki jih bomo zakodirali, in zanjo izračunamo število bitov z enačbo 5.18, ki postane začetna vrednost b_p).

$$b_p = \lceil \log_2(\overline{\sigma}) \rceil \tag{5.18}$$

Koda sestoji iz kontrolnih bitov in bitov, ki zakodirajo število σ_i. Velja:

- $b_a \leq b_p$: v tem primeru je kontrolni bit 0, ki mu sledi binarna predstavitev števila σ_i, zapisana z b_p biti;
- $b_a > b_p$: kontrolne bite predstavlja unarna koda števila $(b_a - b_p)$ (za vejico uporabimo 0), zatem pa zapišemo $b_a - 1$ manj pomembnih bitov števila σ_i (bita MSB ne zapišemo, saj vemo, da je 1).
- Po vsakem koraku kodiranja postane $b_p = b_a$.

$$I = \langle 15\ 6\ 2\ 3\ 0\ 0\ 0\ 0\ 4\ 5\ 1\ 7\ 8 \rangle$$

Primer 5.44: Vhodno zaporedje nenegativnih števil, ki jih bomo zakodirali z BASC

BASC razložimo s primerom 5.44 in postavimo $b_p = 5$. Postopek kodiranja vidimo v razpredelnici 5.17. Razložimo nekaj korakov. Kodirnik inicializiramo s podano vrednostjo $b_p = 5$. Prvo število $\sigma_0 = 15$ bi lahko zapisali z najmanj štirimi biti, zato je $b_a = 4$. Ker je $b_a < b_p$, kodirnik zapiše kontrolni bit ⟨0⟩, ki mu sledi binarni zapis σ_0 z $b = 5$ biti (v razpredelnici 5.17 ločimo kontrolne bite od bitov, ki kodirajo vrednost, z znakom |). Tvorimo ⟨0|01111⟩, b_p pa pridobi vrednost 4. Naslednje število je $\sigma_1 = 6$, $b_a = 3$, dobimo kodo ⟨0|0110⟩ ter postavimo $b_p = 3$. Zatem kodiramo število $\sigma_2 = 2$ kot ⟨0|010⟩ in postavimo $b_p = 2$. $\sigma_3 = 3$ zakodiramo zelo učinkovito kot ⟨0|11⟩, b_p pa ostane 2. Prvo od štirih ničel zakodiramo kot ⟨0|00⟩ in postavimo $b = 0$. Preostale ničle predstavimo vsako s samo po enim kontrolnim bitom ⟨0⟩. Naslednje število je $\sigma_8 = 4$. $b_a = 3 > b_p$, zato kodirnik zapiše $b_a - b_p = 3$ bitov 1, ki jim sledi vejica ⟨0⟩, tej pa preostala manj pomembna bita števila 4, to je ⟨00⟩. Koda je torej ⟨1110|00⟩, b_p pa postavimo na 3. $\sigma_9 = 5$ zapišemo z $b = 3$ biti. Kodirnik torej zapiše ⟨0|101⟩ ter nadaljuje kodiranje do zadnjega števila.

Dekodiranje poteka po obratnem postopku. Zanj potrebujemo začetno vrednost $b_p = 5$ in seveda zaporedje bitov iz primera 5.45, ki jih je zapisal kodirnik.

$$I = \langle 001110011000100110000001110000101000111011110000 \rangle$$

Primer 5.45: Zaporedje bitov, ki jih bomo dekodirali z BASC, ko je $b_p = 5$

Postopek dekodiranja razložimo s pomočjo razpredelnice 5.18. Prvi kontrolni bit je ⟨0⟩. To pomeni, da bomo prebrali naslednjih $b_p = 5$ bitov iz I in jih pretvorili v desetiško vrednost. Dobimo $\sigma_0 = 15$. Za njegov zapis

Razpredelnica 5.17: Kodiranje BASC; začetna vrednost parametra $b_p = 5$

i	σ_i	b_p	b_a	koda
/		5	/	
0	15	5	4	0|01111
1	6	4	3	0|0110
2	2	3	2	0|010
3	3	2	2	0|11
4	0	2	0	0|00
5	0	0	0	0|
6	0	0	0	0|
7	0	0	0	0|
8	4	0	3	1110|00
9	5	3	3	0|101
10	1	3	1	0|001
11	7	1	3	110|11
12	8	3	4	10|000

bi potrebovali samo štiri bite, zato postavimo $b_a = 4$. Naslednjo iteracijo začnemo z $b_p = b_a = 4$ in preberemo naslednji kontrolni bit iz I. Ker je ta $\langle 0 \rangle$, preberemo naslednje 4 bite in dekodiramo $\sigma_1 = 6$. Na ta način postopek nadaljujemo, dokler v deveti vrstici razpredelnice ne zaznamo kontrolni bit $\langle 1 \rangle$. Vrednost p_b inkrementiramo tolikokrat, kot je zaporednih bitov 1. V našem primeru so trije biti 1, zatem sledi zadnji kontrolni bit 0, ki predstavlja vejico. S tem dobimo vrednost $b_a = 3$, iz I pa zatem preberemo samo dva bita, saj vemo, da je bit MSB 1. Na ta način dekodiramo $\sigma_8 = 4$, za zapis katerega potrebujemo 3 bite, zato postavimo $b_a = 3$. Naslednji kontrolni bit je $\langle 0 \rangle$. Preberemo $b_a = 3$ bite in jih dekodiramo. Dobimo $\sigma_9 = 5$, za zapis katerega tudi potrebujemo $b_a = 3$ bite. Tudi naslednji kontrolni bit je $\langle 0 \rangle$. Preberemo 3 bite in dekodiramo vrednost $\sigma_{10} = 1$, za zapis katerega potrebujemo $b_a = 1$ bit. Postavimo $b_p = b_a = 1$. Naslednji kontrolni biti so $\langle 110 \rangle$, s katerimi dvakrat inkrementiramo b_p, ki dobi vrednost 3. Preberemo naslednja bita iz I, ki jima dodamo MSB 1, tako da dekodiramo vrednost $\sigma_{11} = 7$. V bistvu že vemo, da za kodiranje te vrednosti potrebujemo $b_a = 3$ bite, torej postavimo $b_p = 3$. Kontrolna bita sta dva, $\langle 10 \rangle$, zato inkrementiramo b_p na vrednost 4, iz I pa preberemo 3 bite, tako da dekodiramo zadnjo vrednost, ki je $\sigma_{12} = 8$.

BASC lahko izboljšamo z uporabo prisekane binarne kode ali kodiranja

Razpredelnica 5.18: Dekodiranje BASC pri začetni vrednosti $b_p = 5$

O	b_p	σ_i	b_a
001111	5	15	4
00110	4	6	3
0010	3	2	2
011	2	3	2
000	2	0	0
0	0	0	0
0	0	0	0
0	0	0	0
111000	0	4	3
0101	3	5	3
0001	3	1	1
11011	1	7	3
10000	3	8	4

FELICS (spoznali ga bomo v podpoglavju 5.6.5).

5.6.4 Prilagodljivo binarno zaporedno kodiranje z vrnitvijo

BASC je zelo učinkovit, če se vrednosti močno ne spreminjajo ali če se spreminjajo v gručah. V tem primeru izda kodirnik samo en kontrolni bit, to je bit $\langle 0 \rangle$. Izkaže se, da je BASC v tem primeru učinkovitejši od Golombovega kodiranja. Če pa se vrednosti v I močno spreminjajo, na primer pri kodiranju RLE, se uspešnost BASC zmanjša. Rešitev, ki so jo predlagali v [48], težavo premaga z uvedbo globalnega parametra b_r, na katero vrednost se postavi b_p po vsakem koraku kodiranja. Postopek imenujemo BASCRB (angl. Binary Adaptive Sequential Coding with Return to Bias) in je celo preprostejši, kot je izvorni algoritem BASC.

Primer delovanja BASCRB v primerjavi s kodiranjem BASC za zaporedje I iz primera 5.46 vidimo v razpredelnici 5.19. BASCRB je v tem

$$I = \langle 14\ 5\ 4\ 0\ 0\ 18\ 7\ 20\ 4\ 6\ 2\ 5 \rangle$$

Primer 5.46: Vhodno zaporedje za prikaz primernosti BASCRB

primeru zahteval 55 bitov, BASC pa 61. V [48] so z eksperimenti pokazali, da je BASCRB učinkovitejši tudi od Golombovega kodiranja. Parameter

Razpredelnica 5.19: Kodiranje BASC z začetno vrednostjo $b_p = 5$ in kodiranje BASCRB z $b_r = 3$

	BASC			BASCRB		
σ_i	b_p	b_a	bits	b_r	b_a	bits
14	5	4	0|01110	3	4	10|110
5	4	3	0|0101	3	3	0|101
4	3	3	0|100	3	3	0|100
0	3	0	0|000	3	0	0|000
0	0	0	0|	3	0	0|000
18	0	5	111110|0010	3	5	110|0010
7	5	3	0|00111	3	3	0|111
20	3	5	110|0100	3	5	110|0100
4	5	3	0|00100	3	3	0|100
6	3	3	0|110	3	3	0|110
2	3	2	0|010	3	2	0|010
5	2	3	10|01	3	3	0|101

b_r lahko določimo s poskušanjem ali s primerno hevristiko; eno od možnih podaja enačba 5.19.

$$b_r = \left\lceil \log_2 \left(\frac{1}{n} \sum_{i=0}^{n-1} I_i \right) \right\rceil \tag{5.19}$$

5.6.5 Interpolativno kodiranje

Interpolativno kodiranje je predlagal Moffat s sodelavci [49, 50]. Tudi ta metoda posameznim simbolom priredi kode s spremenljivo dolžino VLC, a ta dolžina je dinamična in je lahko v nekaterih primerih celo enaka nič. Posameznemu simbolu namreč ne priredimo enolične kode VLC, ampak se koda spreminja in je odvisna od karakteristik celotnega vhodnega zaporedja. Takšnemu pristopu pravimo **holističen pristop** (holism je grška beseda, ki pomeni popolnost, celoto) [30]. Kodiranje ne poteka tradicionalno zaporedoma znak po znaku od začetka do koncu zaporedja, ampak po vnaprej določenem vrstnem redu. Koda simbola, ki ga kodiramo, je zato precej bolj odvisna od njegovega položaja v zaporedju kot od njegove vrednosti.

Interpolativno kodiranje razložimo s primerom 5.47a, kjer $|I| = 12$, z abecedo $\Sigma = \{a, b, c\}$, $\sigma_i \in \Sigma$. Vsakemu znaku priredimo celo številko, prvi

znak pa dobi vrednost 1. Naj velja: a $= 1$, b $= 2$ in c $= 3$. Dobimo zaporedje N (primer 5.47b), iz tega pa sestavimo zaporedje strogo naraščajočih kumulativnih vrednosti C (primer 5.47c).

(a) $I = \langle$cbbaaacaaccc$\rangle$

(b) $N = \langle$3 2 2 1 1 1 3 1 1 3 3 3$\rangle$

(c) $C = \langle$3 5 7 8 9 10 13 14 15 18 21 24$\rangle$

Primer 5.47: (a) Vhodno zaporedje, (b) simbolom prirejene številke in (c) zaporedje kumulativnih vrednosti

Metoda kodira zaporedje C. Vrstni red kodiranja kumulativnih vrednosti je običajno določen z rekurzivno delitvijo zaporedja C v dve enaki polovici. Za nadzor rekurzije potrebujemo spremenljivki L in H, ki določata spodnji in zgornji indeks trenutno obravnavanega zaporedja. Ko je $L = H$, se rekurzija prekine. Položaj vrednosti, ki jo bomo kodirali, je najlažje izbrati na sredini med L in H, kot določa enačba 5.20.

$$mid = \left\lfloor \frac{L+H}{2} \right\rfloor \tag{5.20}$$

Zatem določimo interval $[m_{lower}, m_{upper}]$, na katerem se nahaja C_{mid}, $C_{mid} \in [m_{lower}, m_{upper}]$. Razmišljajmo takole: vrednost elementa C_L poznamo. Ker je C strogo monotono naraščajoče zaporedje, dobimo najmanjšo možno vrednost za C_{mid} tako, da prištejemo toliko enic k C_L, kot je razdalja med mid in L, s čimer dobimo spodnjo vrednost intervala m_{lower}; izračunamo jo torej z enačbo 5.21:

$$m_{lower} = C_L + (mid - L) \tag{5.21}$$

S podobnim razmislekom določimo tudi zgornjo mejo intervala m_{upper}. Največjo vrednost v C_{mid} bi dobili, če vrednost C_H zmanjšujemo za 1 tolikokrat, kot je razdalja med zgornjo mejo H in mid (enačba 5.22).

$$m_{upper} = C_H - (H - mid) \tag{5.22}$$

S poznavanjem intervala lahko zdaj enostavno zakodiramo vrednost C_{mid}. Število različnih vrednosti, ki jih lahko zavzame C_{mid}, določimo z enačbo 5.23:

$$d = m_{upper} - m_{lower} + 1, \tag{5.23}$$

nato pa izračunamo število bitov u za zapis d možnih vrednosti z že znano enačbo 5.16. Vrednost C_{mid} nato premaknemo na interval $[0, d-1]$, torej $C_{mid} = C_{mid} - m_{lower}$, in jo nazadnje v binarni obliki zapišemo na izhod O z u biti. Interpolativno kodiranje prikažimo na zaporedju kumulativnih vrednosti C iz primera 5.48.

i	0	1	2	3	4	5	6	7	8	9	10	11
$C = \langle$	3	5	7	8	9	10	13	14	15	18	21	24$\rangle$

Primer 5.48: Zaporedje C s pripadajočimi indeksi

Na začetku je $L = 0$ in $H = 11$. Z enačbo 5.20 izračunamo $mid = 5$, z enačbama 5.21 in 5.22 pa $m_{lower} = 3 + (5-0) = 8$ in $m_{upper} = 24 - (11 - 5) = 18$, kar nam določa interval za C_{mid}. Vseh možnih vrednosti za C_{mid} je 11, ki jih moremo zakodirati z $u = 4$ biti (enačba 5.16). Opravimo še premik $C_{mid} = 10 - 8 = 2$, kar kot $\langle 0010 \rangle$ zapišemo v O. Celoten rekurzivni postopek kodiranja prikažemo v razpredelnici 5.20.

Vidimo, da v treh primerih kode nismo potrebovali. Ko sta $L = 2$ in $H = 5$, sta vrednosti $C_2 = 7$ in $C_5 = 10$. Ker je C strogo monotono naraščajoč, vemo, da sta na mestih C_3 in C_4 lahko samo vrednosti 8 in 9. To formalno ugotovimo tako, da preverimo enačbo 5.24.

$$C_H - C_L = H - L \tag{5.24}$$

To bo lahko opravil tudi dekodirnik, ki bo samodejno vstavil vrednosti. Enak primer je tudi, ko sta $L = 6$ in $H = 8$. Nekoliko drugače pa moramo razmišljati, ko je $L = 8$ in $H = 11$. Razlika med $C_{11} = 24$ in $C_8 = 15$ je 9, kar je mnogokratnik dolžine abecede $|\Sigma| = 3$. Edina možnost, da bomo od elementa $C_8 = 15$ dosegli vrednost v $C_{11} = 24$, je, da na prosti mesti C_9 in C_{10} vpišemo vrednosti tako, da vrednosti v C_L dvakrat prištejemo $|\Sigma| = 3$. Tudi ta primer lahko dekodirnik zazna z enačbo 5.25.

$$C_H - C_L = |\Sigma| \cdot (H - L) \tag{5.25}$$

Če enačba velja, vpišemo med dekodiranjem na manjkajoča mesta vrednosti C_L, zaporedoma povečane za $|\Sigma|$. V našem primeru nam je na ta način uspelo 12 celoštevilskih vrednosti zapisati s samo štirinajstimi biti, kot vidimo na primeru 5.49, kar je vsekakor odličen dosežek. Če izračunamo informacijsko entropijo vhodnega zaporedja I iz primera 5.47a, dobimo $H(I) = 1{,}4834$.

Dekodiranje. Za dekodiranje kumulativnih vrednosti potrebujemo dolžino polja $|C|$, spodnjo C_L in zgornjo vrednost C_H ter zaporedje bitov. V

$$O = \langle 00101001001010 \rangle$$

Primer 5.49: Zaporedje bitov pri interpolativnem kodiranju

Razpredelnica 5.20: Prikaz stiskanja z interpolativnim kodiranjem

L	H	mid	$[m_{lower}, m_{upper}]$	$C_{mid} - m_{lower}$	koda
0	11	5	{8, 18}	10-8=2	0010
0	5	2	{5, 7}	7-5=2	10
0	2	1	{4,6}	5-4=1	01
2	5	/	/	/	/
5	11	8	{13, 21}	15-13=2	0010
5	8	6	{11, 13}	13-11=2	10
6	8	/	/	/	/
8	11	/	/	/	/

našem primeru kodirnik alocira zaporedje C dolžine 12 elementov, postavi $L = 0$ in $H = 11$ ter napolni vrednosti $C_L = 3$ in $C_H = 24$. Preostale vrednosti bomo rekonstruirali s pomočjo zaporedja bitov iz primera 5.49 in pravil, ko za kodiranje znakov nismo poslali na izhod nobenega bita (preverjali bomo enačbi 5.24 in 5.25). Stanje dekodirnika po inicializaciji vidimo kaže primer 5.50a.

	i	0	1	2	3	4	5	6	7	8	9	10	11
(a)	$C = \langle$	3											24$\rangle$
(b)	$C = \langle$	3					10						24$\rangle$
(c)	$C = \langle$	3	5	7	8	9	10	13		15			24$\rangle$

Primer 5.50: Postopek dekodiranja

Z enačbo 5.20 najprej izračunamo položaj elementa $mid = 5$, ki ga bomo dekodirali, nato pa z enačbama 5.21 in 5.22 izračunamo $m_{lower} = 8$ in $m_{upper} = 18$, z enačbo 5.23 izračunamo d ter z enačbo 5.16 določimo število bitov u, ki jih bomo prebrali iz zaporedja bitov. V tem primeru imamo $u = 4$, zato preberemo bite $\langle 0010 \rangle = 2$. Vrednosti prištejemo m_{lower} in dobimo $C_{mid=5} = 10$ (glej primer 5.50b).

Dekodiranje nadaljujemo rekurzivno na enak način, kot je potekalo kodiranje. Postavimo $L = 0$ in $H = 5$, izračunamo $mid = 2$, $m_{lower} = 5$ in $m_{upper} = 7$ ter $u = 2$. Preberemo bita $\langle 10 \rangle = 2$ in k rezultatu prištejemo m_{lower}, kar vpišemo v $C_2 = 7$. V naslednjem rekurzivnem klicu imamo

$L = 0$ in $H = 2$, $mid = 1$, $m_{lower} = 4$, $m_{upper} = 6$ ter $u = 2$. Preberemo naslednja 2 bita, to sta $\langle 01 \rangle$, njuno vrednost prištejemo k m_{lower} in vsoto vpišemo v $C_1 = 5$. Sledi rekurzivni klic z $L = 2$ in $H = 5$. Ker dekodirnik z enačbo 5.24 ugotovi, da je $(C_5 = 10 - C_2 = 7) = 3$ enako $H - L = 3$, ne preberemo nobenega bita iz vhodnega zaporedja, saj vemo, da lahko števila med L in H vpišemo tako, da vrednost iz $C_L = 8$ zaporedoma inkrementiramo; inkrementirane vrednosti vpisujemo na položaje od $L+1$ do $H-1$. S tem smo zapolnili levo polovico polja C, dekodiranje desne polovice pa začnemo z $L = 5$ in $H = 11$. Izračunamo $mid = 8$, $m_{lower} = 13$, $m_{upper} = 21$ in dobimo $u = 4$. Preberemo naslednje štiri bite $\langle 0010 \rangle$, jih dekodiramo in dobimo $C_8 = 15$. Nadaljujemo z $L = 5$ in $H = 8$, dobimo $mid = 6$, $m_{lower} = 11$, $m_{upper} = 13$ ter $u = 2$. Zato preberemo naslednja dva bita $\langle 10 \rangle$ in dobimo $C_6 = 13$. Trenutno stanje kaže primer 5.50c.

V naslednjem rekurzivnem klicu sta $L = 6$ in $H = 8$. Ker je enačba 5.25 izpolnjena, nedvoumno vemo, da je $C_7 = 14$. Sledi rekurzivni klic z $L = 8$ in $H = 11$. Tokrat pa velja enačba 5.25, zato napolnimo mesti C_9 in C_{10} tako, da k $C_L = 15$ prištevamo $|\Sigma| = 3$. Dobimo $C_9 = 18$ in $C_{10} = 21$. S tem se rekurzivnimi klici zaključijo, zaporedje pa smo dekodirali. Nazadnje iz C tvorimo zaporedje števil N, iz tega pa vhodno zaporedje I ob poznavanju, kako smo $\sigma_i \in \Sigma$ priredili vrednosti.
Sam postopek interpolativnega kodiranja je možno izvesti preprosteje, tako kot je pokazano v [51]:

- zahtevo po strogo naraščajočem kumulativnem zaporedju C sprostimo;
- izračun vrednosti m_{lower} in m_{upper} ni več nujen, saj je število različnih vrednosti kar $d = C_H - C_L$;
- prvi primer, ko kodirnik ne zapiše nobenega bita, zaznamo še enostavneje. Če je $C_H = C_L$, potem na mesta med $L + 1$ in $H - 1$ zapišemo vrednost iz C_H.

Izboljšavo interpolativnega kodiranja najdemo v [52], kjer se izkaže, da je na nekaterih intervalih bolje zapisati vrednost C_{mid} z Golombovim kodiranjem. V [53] pa je bilo interpolativno kodiranje izboljšano s tako imenovanimi Ψ-kodami in kodiranjem FELICS, ki si ga bomo pogledali tudi v nadaljevanju.

5.6.5.1 FELICS

FELICS (angl. Fast, Efficient, Lossless Image Compression System) so razvili za brezizgubno stiskanje slik z zveznimi barvnimi toni [54]. Vrednost

piksla, ki ga kodiramo, določimo s pomočjo dveh že kodiranih pikslov. Eden od njiju ima manjšo vrednost (v našem primeru je ta označena z m_{lower}), drugi pa večjo (m_{higher}). Kodo, ki jo priredimo kodiranemu pikslu, določimo ob predpostavki, da bo barva kodiranega piksla najverjetneje blizu aritmetične sredine obeh vrednosti. Zato takšnim vrednostim priredimo krajše kode. FELICS obravnava tudi primere, ko je vrednost kodiranega piksla izven območja $[m_{lower}, m_{upper}]$, a to v našem primeru ni možno. Ko smo z enačbo 5.23 določili število vrednosti d na intervalu, določimo kode FELICS na naslednji način: Najprej izračunamo število bitov u_s za tako imenovane kratke kode z enačbo (5.26).

$$u_s = \lfloor \log_2 d \rfloor \tag{5.26}$$

Nato izračunamo števili a in b

$$\begin{aligned} a &= 2^{u_s+1} - d, \\ b &= 2(d - 2^{u_s}), \end{aligned} \tag{5.27}$$

kjer a določa število kratkih kod dolžine u_s bitov, b pa število daljših kod dolžine $(u_s + 1)$.

Razpredelnica 5.21: Kode FELICS

vrednost v C_{mid}	koda
13	0001
14	001
15	010
16	011
17	100
18	101
19	110
20	111
21	0000

Vzemimo primer 5.50, ko je $m_{lower} = 13$, $m_{upper} = 21$ in $d = H - L + 1 = 9$. Potem je $u_s = 3$, število krajših kod $a = 2^4 - 9 = 7$, število daljših pa $b = 2(9 - 2^3) = 2$. Kratke kode so: $8 - 1 = \langle 111 \rangle$, $8 - 2 = \langle 110 \rangle$, do $8 - 7 = \langle 001 \rangle$, daljši kodi pa $\langle 0000 \rangle$ in $\langle 0001 \rangle$. Krajše kode so razporejene na sredini področja, daljše pa na obeh krajiščih. b je sod, zato lahko množico števil vedno razdelimo v dve enako veliki podmnožici. Razpredelnica 5.21

kaže vse kode FELICS za interval $[13, 21]$. Seveda je kode FELICS možno enolično dekodirati. V tem primeru vidimo, da prve tri ničle označujejo kodiranje z daljšimi kodami, če pa imamo enico prej, smo uporabili kodiranje s krajšimi kodami.

V primeru interpolativnega kodiranja iz razpredelnice 5.20 je koda za vrednost $m_8 = 15$ bila $\langle 0010 \rangle$, v načinu FELICS pa bi jo kodirali kot $\langle 010 \rangle$ in prihranili en bit. En bit bi prihranili tudi v primeru, ko na intervalu $[4, 6]$ kodiramo vrednost $m_1 = 5$, ki bi jo zapisali samo z enim bitom, to je $\langle 1 \rangle$. Na ta način bi za kodiranje 12 števil potrebovali samo 12 bitov in dodatno izboljšali učinkovitost kodiranja.

Naloge

1. Vhodno zaporedje je datoteka ASCII s 1.215 znaki. Z algoritmom stiskanja smo datoteko zapisali s 1.142 biti. Izračunajte razmerje stiskanja, faktor stiskanja in prihranek.
2. Kako delimo algoritme stiskanja?
3. Skicirajte cevovod izgubnega stiskanja.
4. S primerom pokažite idejo čelnega stiskanja.
5. Opišite različice stiskanja RLE.
6. Z RLE stisnite zaporedje $I = \langle \mathsf{abdddddabeeeeeaabbb} \rangle$, če je $t = 2$.
7. Izračunajte informacijsko entropijo zaporedij $\langle \mathsf{yyxxxxyyzzxyz} \rangle$ in $\langle \mathsf{uvuvuvuvuvuvuv} \rangle$.
8. Naj bo $I = \langle$Naj viharja moč razsaja, hraste cepi, skale taja.$\rangle$. Izračunajte informacijsko entropijo. Koliko bitov bi potrebovali za kodiranje zaporedja I?
9. Kakšno lastnost imajo predponske kode?
10. Ali so kode $\langle 0 \rangle$, $\langle 10 \rangle$, $\langle 110 \rangle$, $\langle 1110 \rangle$, $\langle 1111 \rangle$ predponske? Odgovor utemeljite.
11. Ali so kode $\langle 0 \rangle$, $\langle 10 \rangle$, $\langle 110 \rangle$, $\langle 010 \rangle$, $\langle 011 \rangle$ predponske? Odgovor utemeljite.

12. Naj bo $I = \langle$kukurukukukokorona$\rangle$, ki je zapisan v razširjenem naboru ASCII. Zanj določite Shannon-Fanojeve kode. Stisnjeno zaporedje zapišite v bitno zaporedje O. Kakšen je prihranek?

13. Razširite bitno zaporedje iz naloge 12. Katere podatke, poleg zaporedja O, še potrebujete?

14. Kaj je varianca kode? Kolikšno varianco kode imata kodi $\langle$011$\rangle$ in $\langle$000111$\rangle$?

15. Kakšno pravilo uporabimo pri Huffmanovem algoritmu, da v primeru dvoumnosti sestavimo enolično kodirno-dekodirno drevo?

16. Zaporedje iz naloge 12 stisnite s Huffmanovim kodiranjem. Kakšno razmerje stiskanja dosežete, če je I zapisan v osnovnem naboru ASCII?

17. Razširite stisnjeno zaporedje iz prejšnje naloge.

18. Zaporedje $I = \langle$dedek$\langle$EOF$\rangle\rangle$ zakodirajte s Huffmanovim algoritmom s prilagajanjem.

19. Stisnjeno zaporedje iz prejšnje naloge razširite.

20. Za sporočilo $I = \langle$perunperunaperunuperunapriperunusperunom$\rangle$ izračunajte informacijsko entropijo. Zaporedje nato stisnite s Huffmanovim algoritmom in aproksimativnim Huffmanovim algoritmom s prilagajanjem. Kateri algoritem se izkaže bolje, če upoštevate dolžino glave ali če dolžine glave ne upoštevate?

21. Pravilnost izhodnih zaporedij iz prejšnje naloge preverite tako, da jih razširite.

22. S primerom $I = \langle$svetjelep$\rangle$ razložite idejo aritmetičnega kodiranja.

23. Zaporedje $I = \langle$hagavaga$\rangle$ zakodirajte s celoštevilsko implementacijo aritmetičnega kodiranja s pomikanjem. Začetne vrednosti za $mLow$ in $mHigh$ postavite na 00000 in 99999. Pravilnost preverite z dekodiranjem.

24. Kdaj nastane nevarnost podkoračitve pri celoštevilski implementaciji aritmetičnega kodiranja s pomikanjem? Kakšen je postopek, da se iz nevarnosti rešimo? Razložite s primerom.

25. Razložite aritmetično kodiranje s skaliranjem.

26. Z aritmetičnim kodiranjem s skaliranjem zakodirajte zaporedje $I = \langle \mathsf{babica} \rangle$, če je dolžina registra $r = 8$. Rezultat zapišite v zaporedje O.

27. Zaporedje O iz prejšnje naloge razširite.

28. Ob primeru naloge 20 razložite algoritem LZ77.

29. Z algoritmom LZSS zakodirajte sporočilo I iz naloge 20, pri čemer naj bo velikost slovarja 16 zlogov, velikost primerjalnega pomnilnika pa 6 zlogov. Razmislite o primernem načinu kodiranja žetonov. Rezultat zapišite v zaporedje O.

30. O iz prejšnje naloge razširite.

31. Z algoritmom LZ78 zakodirajte zaporedje $I = \langle \mathsf{gor\ gornji\ gornja\ gorjanci} \rangle$ in rezultat zapišite v izhodno zaporedje O.

32. Zaporedje iz prejšnje naloge dekodirajte.

33. Zaporedje I iz naloge 31 zakodirajte z algoritmom LZW, pri čemer Σ sestoji iz malih tiskanih črk slovenske abecede. Izhodno zaporedje dekodirajte.

34. Z Golombovim kodiranjem zakodirajte števila 22, 27 in 34, če je Golombov parameter $k = 5$. Zakodirajte še števila 15, −12, 4, −9, če je $k = 6$.

35. Napišite program, ki bo dano število ob podanem parametru zakodiral z Golombovim kodiranjem. Zatem napišite program za dekodiranje Golombove kode.

36. Določite Golomb-Riceove kode za števila 18, −27, −3, 8, če je parameter $k = 4$.

37. Zaporedje I iz naloge 31 zakodirajte z Golombovim kodiranjem.

38. Z metodo binarnega zaporednega kodiranja (BASC) zakodirajte zaporedje števil $I = \langle 5, 12, 13, 9, 0, 0, 0, 24, 6, 25, 5, 10 \rangle$. Začetna vrednost $b_p = 0$. Pravilnost preverite z dekodiranjem.

39. Zaporedje iz naloge 38 zakodirajte z metodo BASCRB, pri čemer določite parameter b_p s hevristiko. Tudi tokrat opravite dekodiranje.

40. Kako bi s kodiranjem BASC in BASCRB zakodirali tako negativna kot pozitivna cela števila?

41. Primer 5.47 zakodirajte z interpolativnim kodiranjem in kodiranjem FELICS. Vam je dvanajst števil uspelo zapisati samo z dvanajstimi biti?

42. Zaporedje $I = \langle$cccdabcccaaa$\rangle$ zakodirajte z interpolativnim kodiranjem. Razmislite, kako bi bilo najugodneje števila prirediti znakom? Zaporedje bitov tudi razširite.

43. Zaporedje iz naloge 42 zakodirajte s Huffmanovim kodiranjem. Katero kodiranje je uspešnejše?

44. Naj bo zaporedje $I = \langle$brmbrmbrrrrrrm$\rangle$, ki ga želimo zakodirati z interpolativnim kodiranjem, pri čemer za zapis vrednosti C_m uporabite kodiranje FELICS. Ali je v tem primeru bila uporaba kodiranja FELICS upravičena?

Poglavje 6

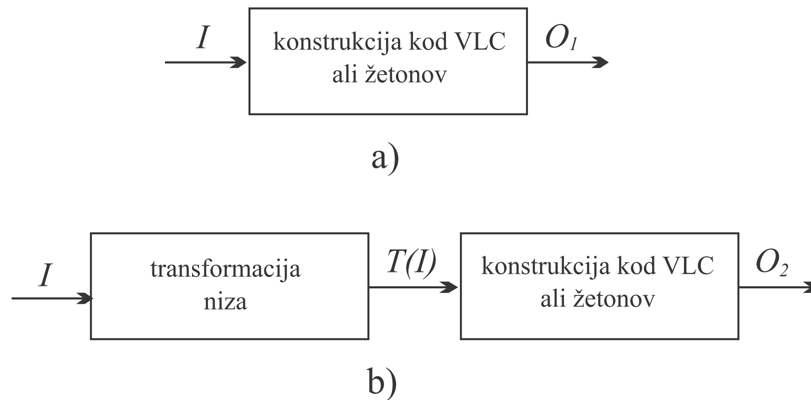
Metode transformacije zaporedij

Metode brezizgubnega stiskanja podatkov, ki smo jih spoznali v poglavju 5, poskušajo vhodno zaporedje I zapisati krajše s kodami s spremenljivo dolžino VLC ali z žetoni (slika 6.1a). Metode so doživele množico izboljšav, s čimer so bile dosežene skrajne zmožnosti teh pristopov. A vedno se najde še kakšna dobra misel, dobra ideja. Kaj, če bi poskušali preoblikovati zaporedje I s primerno transformacijo T tako, da bi $T(I)$ imel manjšo informacijsko entropijo (glej podpoglavje 5.2) in bil tako bolj stisljiv (slika 6.1b)? Želeli bi si torej, da bi veljalo $H(I) > H(T(I))$ in $|O_1| > |O_2|$. Takšne transformacije dejansko obstajajo in niso namenjene samo stiskanju podatkov, ampak je njihova uporaba zelo široka. Nekaj takšnih transformacij si bomo pogledali v nadaljevanju.

6.1 Transformacija premik naprej

Ena izmed najpreprostejših in najpogostejših transformacij je transformacija **pomik naprej** (angl. Move-To-Front, MTF), ki jo je predlagal Ryabko [55]. Pogosto se uporablja pri stiskanju podatkov, pa tudi kot ena od strategij upravljanja s pomnilnikom. MTF pretvori vhodno zaporedje $I = \langle \sigma_i \rangle$, $\sigma_i \in \Sigma$, v izhodno zaporedje $O = \langle o_i \rangle$, kjer je $o_i \in \mathbb{N}_0 = \{0, 1, 2 \cdots |\Sigma| - 1\}$, pri čemer $|I| = |O|$. MTF v bistvu preslika domeno znakov v domeno nenegativnih celih števil. Pri tej preslikavi se *lahko* informacijska entropija zmanjša.

MTF uporablja seznam L, ki ga na začetku napolnimo s simboli $\sigma_i \in \Sigma$. Simbole $\sigma_i \in I$, $i = \{0, 1, 2, \cdots, |I - 1|\}$, nato obdelujemo zaporedno. V



Slika 6.1: Tranformacije zaporedij lahko povečajo učinkovitost stiskanja

seznamu L poiščemo simbol σ_i in njegov položaj (indeks) i pošljemo na izhod O. Zatem položaje simbolov v L med $[0, i-1]$ za eno mesto povečamo, na prvo mesto v seznamu pa zapišemo σ_i, to je, σ_i **premaknemo naprej**.

Zaporedje in abecedo, ki ju bomo uporabili za prikaz delovanja transformacije MTF, kaže primer 6.1.

$$I = \langle \texttt{addddadacbdacbd} \rangle$$

$$\Sigma = \{\texttt{a, b, c, d}\}$$

Primer 6.1: Vhodno zaporedje za transformacijo MTF

Najprej inicializiramo L z znaki abecede, torej $L = \langle \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d} \rangle$ (glej razpredelnico 6.1). Iz I vzamemo prvi simbol $I_0 = \mathtt{a}$ in ga poiščemo v L. Ker je $L[0] = I_0$, pošljemo na izhod indeks 0, status L pa ostane nespremenjen. Naslednji znak, $I_1 = \mathtt{d}$, se v L nahaja na položaju 3, ki ga pošljemo v O. Posodobiti moramo še vsebino L. Vse elemente premaknemo za eno mesto v desno, d pa postavimo na prvo mesto v seznamu. Naslednji simbol $I_2 = \mathtt{d}$ je prvi v seznamu, na izhod pošljemo 0, seznam L pa ostane nespremenjen. Algoritem tako nadaljuje do zadnjega elementa v I, ko dobimo končno transformirano zaporedje, ki ga kaže primer 6.2.

$$O = \langle \texttt{0 3 0 0 0 1 1 1 3 3 3 3 3 3 3} \rangle$$

Primer 6.2: Rezultat transformacije pomik naprej

Hitro lahko ugotovimo, da transformacija MTF zaporedje enakih simbo-

Razpredelnica 6.1: Primer transformacije MTF

I	O	L
/	/	$\langle$a, b, c, d$\rangle$
a	0	$\langle$a, b, c, d$\rangle$
d	3	$\langle$d, a, b, c$\rangle$
d	0	$\langle$d, a, b, c$\rangle$
d	0	$\langle$d, a, b, c$\rangle$
d	0	$\langle$d, a, b, c$\rangle$
a	1	$\langle$a, d, b, c$\rangle$
d	1	$\langle$d, a, b, c$\rangle$
a	1	$\langle$a, d, b, c$\rangle$
c	3	$\langle$c, a, d, b$\rangle$
b	3	$\langle$b, c, a, d$\rangle$
d	3	$\langle$d, b, c, a$\rangle$
a	3	$\langle$a, d, b, c$\rangle$
c	3	$\langle$c, a, d, b$\rangle$
b	3	$\langle$b, c, a, d$\rangle$
d	3	$\langle$d, b, c, a$\rangle$

lov transformira v zaporedje indeksov 0, zaporedje izmenjujočih se simbolov v zaporedje indeksov 1, zaporedje ponavljajočih se parov se preslika v zaporedje indeksov 2 itn., kar lahko vpliva na informacijsko entropijo. V našem primeru velja, da je $H(I) = 1{,}7968$, $H(O) = 1{,}4566$. V nekaterih primerih lahko večkratna zaporedna uporaba MTF dodatno zmanjša informacijsko entropijo [56]. Če bi MTF uporabili nad O, pri čemer bi indekse zaporedja O pretvorili v znake novega zaporedja I, bi se informacijska entropija dejansko zmanjšala še bolj in bi bila le še 1,2366, kar lahko bralec tudi preveri.

Da bi vzpostavili izvorne podatke, potrebujemo tudi inverzni postopek, to je **inverzno transformacijo pomik naprej** (angl. Inverse Move-To-Front, IMTF). Algoritem sprejme na vhodu zaporedje indeksov I, potrebuje pa tudi abecedo Σ. V vsakem koraku algoritem iz L pošlje na izhod simbol σ_i na položaju $L[I_i]$, nato pa premakne σ_i na prvo mesto v L. IMTF za naš primer vidimo v razpredelnici 6.2.

Transformacija premik naprej (angl. Move-To-Front, MTF) je ena od **samoorganizirajočih se podatkovnih struktur**. V literaturi najdemo različne strategije, kako organizirati seznam glede na vhodne podatke, da bo pričakovan dostop do iskanega podatka čim krajši. Poleg MTF velja omeniti

Razpredelnica 6.2: Inverzna transformacija MTF

I_i	σ_i	L
/	/	⟨a, b, c, d⟩
0	a	⟨a, b, c, d⟩
3	d	⟨d, a, b, c⟩
0	d	⟨d, a, b, c⟩
0	d	⟨d, a, b, c⟩
0	d	⟨d, a, b, c⟩
1	a	⟨a, d, b, c⟩
1	d	⟨d, a, b, c⟩
1	a	⟨a, d, b, c⟩
3	c	⟨c, a, d, b⟩
3	b	⟨b, c, a, d⟩
3	d	⟨d, b, c, a⟩
3	a	⟨a, d, b, c⟩
3	c	⟨c, a, d, b⟩
3	b	⟨b, c, a, d⟩
3	d	⟨d, b, c, a⟩

še strategiji **premik za eno mesto** (angl. transponse), ko simbol v L zamenjamo z njegovim predhodnikom, in **premik glede na frekvence** (angl. frequency count), kjer so simboli σ_i v L urejeni glede na pogostost pojavitve do zdaj videnih simbolov. Pregled strategij z njihovo analizo najdemo v [57].

6.2 Transformacija inverzne frekvence

Arnavut in Magliveras [58] sta predstavila algoritem transformacije zaporedij, ki sta ga poimenovala **inverzne frekvence** (angl. inverse frequencies, IF). Algoritem sprejme na vhodu abecedo Σ ter zaporedje $I = \langle \sigma_i \rangle, \sigma_i \in \Sigma$. Tako kot MTF, tudi IF pretvori I v izhodno zaporedje $O = \langle o_i \rangle$, $o_i \in \mathbb{N}_0 = \{0, 1, 2 \cdots |I| - 1\}$, torej tudi IF opravi transformacijo iz domene znakov v domeno nenegativnih celih števil, pri čemer pa je tokrat zaloga vrednosti omejena z dolžino polja I in ne, tako kot pri MTF, s številom simbolov v abecedi Σ.

Algoritem IF vzame simbole σ_i po vrsti iz vhodnega zaporedja I. Za vsak znak najprej zapišemo položaj njegove prve pojavitve v I, za vse naslednje njegove pojavitve pa določimo odmik od predhodnih pojavitev, pri tem pa

preskočimo druge, že uporabljene znake iz Σ. Rezultat shranimo v pomožna zaporedja M_j, $j = 0, 1, 2, \cdots, |\Sigma| - 1$, te pa po vrsti zlepimo v rezultat O. Oglejmo si primer 6.3, ki kaže vhodno zaporedje in njegovo abecedo.

$$I = \langle \texttt{adaadadacbdacbd} \rangle$$

$$\Sigma = \{\texttt{a b c d}\}$$

Primer 6.3: Vhodno zaporedje in abeceda za transformacijo IF

Transformacijo IF določimo v naslednjih štirih korakih:

- $O = \langle\rangle$
- $\sigma_0 = \mathsf{a}$: $M_0 = \langle 0\ 1\ 0\ 1\ 1\ 3 \rangle$; $O = O + M_0$;
- $\sigma_1 = \mathsf{b}$: $M_1 = \langle 9\ 2 \rangle$; $O = O + M_1$;
- $\sigma_2 = \mathsf{c}$: $M_2 = \langle 8\ 1 \rangle$; $O = O + M_2$;
- $\sigma_3 = \mathsf{d}$: $M_3 = \langle 1\ 0\ 0\ 0\ 0 \rangle$; $O = O + M_3$.

Razložimo. Prvi znak abecede $\sigma_0 = \mathsf{a}$. V I je prvi a na položaju 0, kar vpišemo v pomožno zaporedje M_0. Do naslednjega znaka a nas loči en znak, zato je naslednji vnos v M_0 1. Tretji a v I takoj sledi predhodniku, zato je tretji vnos v M_0 0. Do naslednjega znaka a pridemo po enem znaku d, zato vstavimo v M_0 razdaljo 1. Tudi do naslednjega a je razdalja 1. Do zadnjega znaka a nas ločijo trije znaki, zato je zadnja vrednost, vpisana v M_0, 3. Zatem uporabimo drugi simbol iz Σ, to je b. Prvi b v I najdemo šele na položaju 9. Razdalja do naslednjega b je 2, saj že obiskan znak a na položaju I_{11} preskočimo. Naslednji znak iz Σ je c. Prvi c najdemo na položaju 8, kar vpišemo v M_2. Razdalja do drugega znaka c je 1, saj smo znaka b in a že uporabili. Zadnji znak iz Σ je d. Prvi d najdemo na položaju 1, do vseh preostalih znakov d pa je razdalja 0, saj so drugi znaki bili že obiskani. Rezultat transformacije IF kaže primer 6.4a.

(a) $O = \langle 0\ 1\ 0\ 1\ 1\ 3\ 9\ 2\ 8\ 1\ 1\ 0\ 0\ 0\ 0 \rangle$

(b) $O = \langle 0\ 1\ 0\ 1\ 1\ 3\ 9\ 2\ 8\ 1 \rangle$

Primer 6.4: (a) Rezultat transformacije IF in (b) rezultat, ki še omogoča rekonstrukcijo

Iz primera ugotovimo, da se bodo relativne razdalje v pomožnih zaporedjih M_i z vsakim novim simbolom $\sigma_i \in \Sigma$ najverjetneje zmanjševale. Pri zadnjem simbolu so vse enake 0, kar je za stiskanje podatkov zelo dobra novica. Še več, če poznamo dolžino $n = |I|$, polja $M_{|\Sigma|-1}$ sploh ni treba zapisati. Transformacija IF je v tem primeru tista, ki jo kaže primer 6.4b. Vidimo, da bi bilo smiselno oblikovati Σ tako, da bi bil zadnji znak abecede znak z največ pojavitvami.

Da bi lahko izvedli dekodiranje, moramo poleg O inverzni frekvenčni transformaciji (IIF) posredovati dodatne podatke. Dekodirniku moramo namreč sporočiti, kateri indeksi pripadajo posameznim simbolom iz Σ. To lahko storimo na dva načina:

- uvedemo dodatno polje frekvenc F, v katerega vpišemo število pojavitev znakov $\sigma_i \in \Sigma$, ali

- uvedemo stražarja $\sigma_s \notin \Sigma$, ki ga vpišemo v polje O za vsakim pomožnim polje M_{σ_i} razen za zadnjim.

Če kot vhodni podatek za dekodirnik podamo tudi dolžino vhodnega zaporedja $|I|$, zadnjega pomožnega polja $M_{|\Sigma|-1}$ ne potrebujemo.

Najprej uporabimo prvo možnost za dekodiranje našega primera. Dekodirnik prejme podatke iz primera 6.5.

(a) $F = \langle 6\ 2\ 2\ 5 \}$
(b) $\Sigma = \{ \mathtt{a\ b\ c\ d} \rangle$
(c) $I\ = \langle 0\ 1\ 0\ 1\ 1\ 3\ 9\ 2\ 8\ 1\ 1\ 0\ 0\ 0\ 0 \rangle$

Primer 6.5: IIF: (a) frekvence simbolov, (b) abeceda, (c) vhodno zaporedje

Dekodirnik najprej sešteje vse vrednosti v F, s čimer dobimo dolžino $n = 15$ izhodnega zaporedja O, ki ga inicializiramo tako, kot kaže primer 6.6a. V O bomo najprej vstavili prvi znak abecede a. Vemo, da je znakov a šest, ker je $F_0 = 6$. Prvo število v $I_0 = 0$, kar pomeni, da bomo zapisali prvi a na O_0. Naslednja vrednost $I_1 = 1$, zato eno mesto v O preskočimo in na položaj O_2 vpišemo a. Naslednji element $I_3 = 0$, zato zapišemo nov a za pravkar vstavljenim. Ko preberemo prvih 6 vrednosti iz I, je stanje v O takšno, kot kaže primer 6.6b.

Naslednji element v $F = 2$, kar pomeni, da bomo vpisali dva znaka b. $I_6 = 9$ označuje položaj prvega b, naslednji b pa je za dve prazni mesti odmaknjen v desno. Rezultat kaže primer 6.6c. Naslednji element v F podaja informacije za položaje znakov c. Prvi c je na položaju O_8, naslednji pa je za eno prazno mesto odmaknjen v desno (primer 6.6d). Zadnji element

	i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
(a)	$O = \langle$	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_ $\rangle$
(b)	$O = \langle$	a	_	a	a	_	a	_	a	_	_	_	a	_	_	_ $\rangle$
(c)	$O = \langle$	a	_	a	a	_	a	_	a	_	b	_	a	_	b	_ $\rangle$
(d)	$O = \langle$	a	_	a	a	_	a	_	a	c	b	_	a	c	b	_ $\rangle$
(e)	$O = \langle$	a	d	a	a	d	a	d	a	c	b	d	a	c	b	d $\rangle$

Primer 6.6: Postopek inverzne transformacije IF: (a) inicializacija, (b) po vstavitvi prvega znaka abecede, (c) po vstavitvi drugega znaka, (d) po vstavitvi tretjega znaka, (e) zadnji znak abecede vstavimo na prazna mesta

v F pove, da na preostala mesta vpišemo zadnji znak abecede d in dobimo rekonstruiramo zaporedje, (glej primer 6.6e).

Oglejmo si še drugo možnost. Stražar naj bo $\sigma_s =!$. Vhodno zaporedje I ustrezno razširimo, kot kaže primer 6.7. Postopek dekodiranja je enak kot pri prejšnjem primeru, le da takrat, ko zaznamo stražarja, vzamemo naslednji znak iz abecede ter prvi znak za stražarjem interpretiramo kot absolutni položaj prve pojavitve simbola v O.

(a) $n = 15$
(b) $\Sigma = \{$a b c d$\rangle$
(c) $I = \langle$0 1 0 1 1 3 ! 9 2 ! 8 1$\rangle$

Primer 6.7: Vhodni podatki za inverzno transformacijo IF, ko uporabimo stražarja: (a) dolžina zaporedja, (b) abeceda, (c) vhodno zaporedje

Glede na eksperimente, ki sta jih opravila avtorja transformacije IF [59], je IF pri večjih datotekah primernejša za stiskanje podatkov kot transformacija MTF.

6.3 Transformacija kodiranje razdalj

Transformacijo kodiranje razdalj (angl. distance coding, DC) je predlagal Binder [60, 61]. Gre za poenostavljeno transformacijo IF, kjer znake, ki smo jih že srečali, ne preskočimo. Za primer 6.3 bi transformacijo DC določili z naslednjim postopkom:

- $O = \langle\rangle$
- $\sigma_0 =$ a: $M_0 = \langle$0 1 0 1 1 3$\rangle$; $O = O + M_0$;

- $\sigma_1 = \mathsf{b}$: $M_1 = \langle 9\ 3\rangle$; $O = O + M_1$;
- $\sigma_2 = \mathsf{c}$: $M_2 = \langle 8\ 3\rangle$; $O = O + M_2$;
- $\sigma_3 = \mathsf{d}$: $M_3 = \langle 1\ 2\ 1\ 2\ 3\rangle$; $O = O + M_3$.

Rezultat kaže primer 6.8. Rekonstrukcijo opravimo na enak način kot pri transformaciji IF.

$$O = \langle \mathtt{0\ 1\ 0\ 1\ 1\ 3\ 9\ 3\ 8\ 3\ 1\ 2\ 1\ 2\ 3} \rangle$$

Primer 6.8: Rezultat transformacije DC

6.4 Transformacija desno manjše

Tudi to transformacijo je predlagal Arnavut [59]. Naj bo zaporedje $I = \langle \sigma_i \rangle, \sigma_i \in \Sigma$, kjer je Σ abeceda. Transformacijo **desno manjše** (angl. right smaller, RS) dobimo tako, da za vsak znak $\sigma_i \in I$, $0 \leq i < |I|-1$, ugotovimo, koliko znakov σ_j, $i < j \leq |I|-1$, je glede na abecedni vrstni red manjših od znaka σ_i.

(a) $I = \langle \mathtt{caebd} \rangle$
(b) $\Sigma = \{ \mathtt{a\ b\ c\ d\ e} \rangle$

(c) $O = \langle \mathtt{2\ 0\ 2\ 0\ 0} \rangle$

Primer 6.9: Transformacija RS: (a) vhodno zaporedje, (b) abeceda, (c) izhodno zaporedje

Transformacijo RS pojasnimo s primerom 6.9. Transformacija RS vrne izhodno zaporedje O, ki ga kaže primer 6.9c. Razložimo, kako smo dobili O. $I_0 = \mathsf{c}$. Ker sta $I_1 = \mathsf{a}$ in $I_3 = \mathsf{b}$ po abecednem vrstnem redu manjša od c, je vrednost $O_0 = 2$. Sledi $I_1 = \mathsf{a}$. Ker v desno ne najdemo nobenega elementa, ki bi bil po abecednem vrstnem redu pred a, je $O_1 = 0$. Naslednji simbol $I_2 = \mathsf{e}$, desno od njega sta dva manjša elementa glede na abecedni vrstni red, zato je $O_2 = 2$. Ostala sta še dva elementa. Za predzadnjega velja, da je desno od njega večji element, $\mathsf{d} > \mathsf{b}$, zato je $O_3 = 0$. Zadnji element pa na desni nima več nikogar, zato postavimo $O_4 = 0$.

Arnavut [59] je idejo transformacije RS realiziral s transformacijo MTF. Znake iz abecede najprej vstavimo v seznam L. Med postopkom transformacije vzamemo znak $\sigma_i \in I$, $0 \leq i < |I|$, nato ga poiščemo v L. Na

izhod O pošljemo njegov indeks, σ_i pa iz L odstranimo. Zaporedje indeksov predstavlja transformacijo RS, postopek pa vidimo v razpredelnici 6.3.

Razpredelnica 6.3: Transformacija RS z uporabo ideje MTF

I	O	L
/	/	$\langle$a, b, c, d, e$\rangle$
c	2	$\langle$a, b, d, e$\rangle$
a	0	$\langle$b, d, e$\rangle$
e	2	$\langle$b, d$\rangle$
b	0	$\langle$d$\rangle$
d	0	$\langle\rangle$

Pri inverzni transformaciji desno manjše (IRS) najprej napolnimo seznam L z znaki abecede. Vhod v algoritem postane izhod transformacije RS, torej $I = \langle 2\ 0\ 2\ 0\ 0 \rangle$. Elemente iz I jemljemo po vrsti. $I_0 = 2$ kaže na znak c v L, ki ga zapišemo v O, iz L pa c odstranimo. Celotno rekonstrukcijo vidimo v razpredelnici 6.4.

Razpredelnica 6.4: Postopek dekodiranja desno manjše

σ_i	O	L
/	/	$\langle$a, b, c, d, e$\rangle$
2	c	$\langle$a, b, d, e$\rangle$
0	a	$\langle$b, d, e$\rangle$
2	e	$\langle$b, d$\rangle$
0	b	$\langle$d$\rangle$
0	d	$\langle\rangle$

Simboli abecede Σ pa se v I praviloma pojavljajo večkrat, kot kaže primer 6.10. Vidimo, da se znaka a in c v I pojavita dvakrat, kar v abecedi ustrezno označimo, na primer $\Sigma = \{a^2\ b^1\ c^2\}$, lahko pa bi uvedli tudi polje frekvenc F, kot smo pri primeru transformacije IF.

Seznam L inicializiramo tako, da upoštevamo število ponovitev znakov. Postopek transformacije RS z uporabo ideje MTF, ko se znaki lahko ponavljajo, kaže razpredelnica 6.5. Inverzna transformacija IRB deluje na popolnoma enak način kot pri primeru, prikazanem v razpredelnici 6.4.

Seveda imamo takoj ideje za podobne transformacije, kot je transformacija RS. Lahko bi iskali desne elemente, ki so večji (angl. right bigger, RB),

(a) $I = \langle \texttt{acacb} \rangle$
(b) $\Sigma = \{\texttt{a}^2 \ \texttt{b}^1 \ \texttt{c}^2\rangle$

(c) $O = \langle \texttt{0 2 0 1 0} \rangle$

Primer 6.10: Vhodni podatki za transformacijo RS, ko se znaki ponovijo večkrat: (a) vhodno zaporedje, (b) abeceda, (c) rezultat

Razpredelnica 6.5: Transformacija RS, kjer se znaki abecede v I lahko pojavijo večkrat

I	O	L
/	/	⟨a, a, b, c, c⟩
a	0	⟨a, b, c, c⟩
c	2	⟨a, b, c⟩
a	0	⟨b, c⟩
c	1	⟨b⟩
b	0	⟨⟩

ali leve elemente, ki so manjši (angl. left smaller, LS) [59], in še marsikaj.

6.5 Transformacija z drevesom valčkov

Drevo valčkov (angl. wavelet tree, WT) so leta 2003 predstavili Grossi, Gupta in Vitter [62]. Transformacija z drevesom valčkov (angl. wavelet tree transform, WTT) pretvori zaporedje $I = \langle \sigma_i \rangle$, $\sigma_i \in \Sigma$ v bitno zaporedje $O = \langle \beta_i \rangle$, $\beta_i \in \{0, 1\}$. WT je uravnoteženo binarno drevo, ki ga zgradimo *od zgoraj navzdol* z naslednjim postopkom:

1. Σ razdelimo na dve polovici; v Σ_1 in Σ_2 (če je $|\Sigma|$ lih, je $|\Sigma_1| < |\Sigma_2|$).

2. Simbolom iz Σ_1 dodelimo bit 0, simbolom iz Σ_2 pa bit 1.

3. I razdelimo na dve zaporedji, I_1 in I_2, glede na to, v katero abecedo (Σ_1 ali Σ_2) spadajo posamezni simboli σ_i.

4. Σ_1 in I_1 dodelimo levemu potomcu drevesa valčkov, Σ_2 in I_2 pa desnemu.

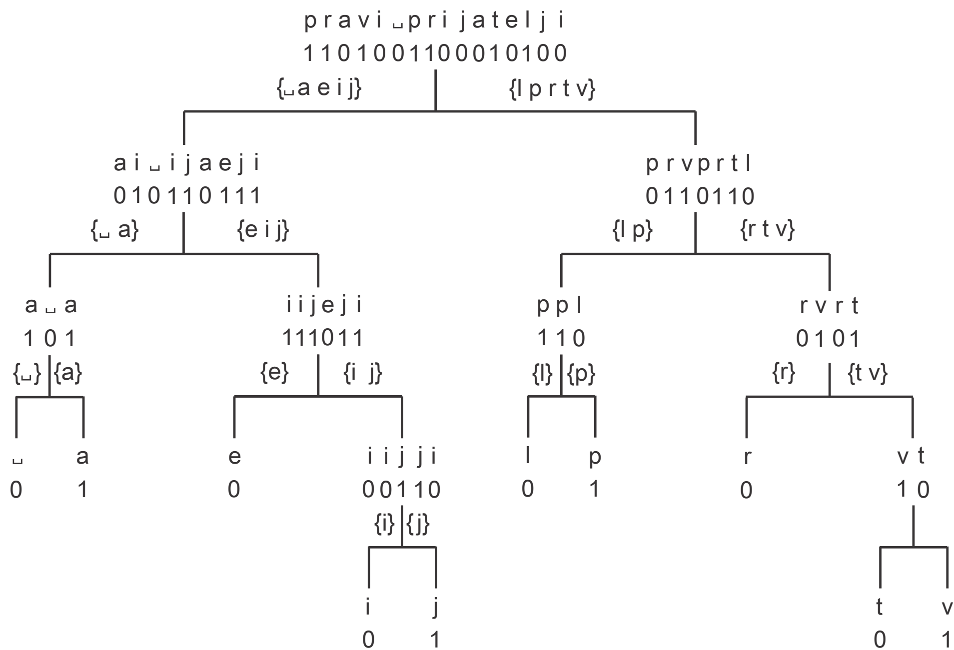
5. Rekurzivno ponavljamo postopek deljenja, dokler je $|\Sigma_i| > 1$.

Postopek tvorbe WT razložimo za podatke iz primera 6.11, pri čemer $n = |I| = 16$ in $|\Sigma| = 10$. Najprej razdelimo Σ. Dobimo $\Sigma_1 = \{$␣ a e i j$\}$ in

(a) $\Sigma = \{$␣ a e i j l p r t v$\}$
(b) $I = \langle$pravi␣prijatelji$\rangle$

Primer 6.11: (a) Abeceda in (b) vhodno zaporedje za WTT

$\Sigma_2 = \{$l p r t v$\}$. Znakom iz abecede Σ_1 pripišemo bit 0, znakom iz Σ_2 pa bit 1 (glej sliko 6.2). Glede na vrednost bita potem razdelimo tudi I na dve podzaporedji $I_1 = \langle$ai␣ ijaeji$\rangle$ in $I_2 = \langle$prvprtl$\rangle$ (v splošnem $|I_1| \neq |I_2|$). Nato rekurzivno nadaljujemo po istem postopku. Σ_1 razdelimo v $\Sigma_{1,1} = \{$␣ a$\}$ in $\Sigma_{1,2} = \{$e i j$\}$. Znakom $\sigma_i \in I_1$ dodelimo ustrezne vrednosti bitov ter razdelimo zaporedje I_1 v zaporedji $I_{1,1} = \langle$a␣ a$\rangle$ in $I_{1,2} = \langle$iijeji$\rangle$. Na enak način obdelamo tudi I_2. Delitev nadaljujemo, dokler v abecedi ne ostane samo en znak. Postopek konstrukcije drevesa valčkov kaže slika 6.2.



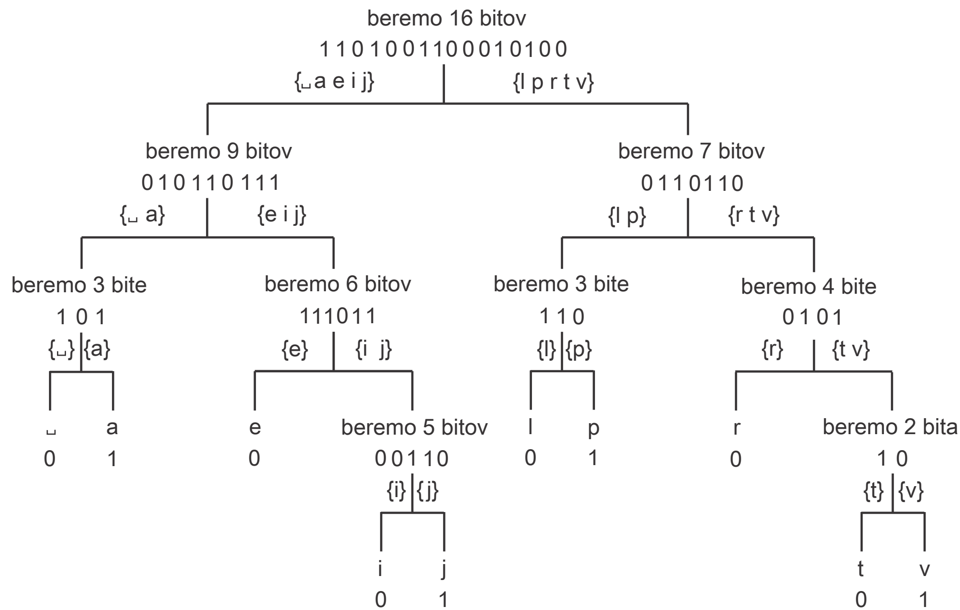
Slika 6.2: Drevo valjčkov.

Izhodno zaporedje O, ki ga kaže primer 6.12, dobimo z izpisom bitov po nivojih od zgoraj navzdol, pri čemer bitov v listih WT ne vključimo.

Za rekonstrukcijo izvornega zaporedja potrebujemo poleg zaporedja bitov I še abecedo Σ in dolžino sporočila $n = |I|$. Za samo dekodiranje drevesa

$$O = \langle \texttt{1101001100010100010110111011011010111101111001010011010} \rangle$$

Primer 6.12: Rezultat WTT – zaporedje bitov



Slika 6.3: Rekonstruirano drevo valčkov

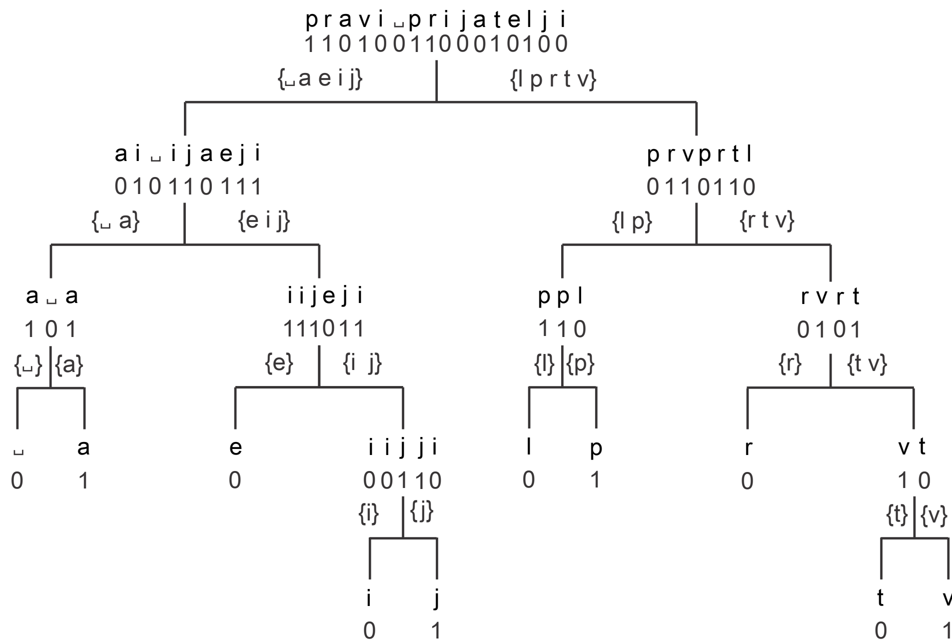
valčkov sicer ni treba konstruirati, vendar je postopek rekonstrukcije lažje spremljati, če gradimo tudi drevo. Rekonstrukcija poteka v dveh korakih:

1. S pomočjo zaporedja bitov zgradimo WT in določimo abecede na posameznih vejah s postopkom *od zgoraj navzdol*.

2. S postopkom *od spodaj navzgor* rekonstruiramo zaporedje.

Najprej razpolovimo vhodno abecedo Σ v Σ_1 in Σ_2, ki ju priredimo levi in desni veji WT. Ker je $n = 16$, preberemo prvih 16 bitov iz I. Ugotovimo, da je med njimi 9 bitov 0 in 7 bitov 1. Biti 0 ustrezajo znakom iz Σ_1, biti 1 pa iz Σ_2. Tako vemo, da naslednjih 9 bitov pripada levemu poddrevesu, 7 bitov, ki sledi, pa desnemu (glej sliko 6.3). Postopek ponovimo za vsak nivo. Razdelimo Σ_1 v $\Sigma_{1,1}$ in $\Sigma_{1,2}$. Ker so v vozlišču WT trije biti 0, vemo, da moramo prebrati 3 bite za levo vejo in 6 bitov za desno vejo, kar ustreza bitom 1. Postopek ponavljamo, dokler $|\Sigma_i| > 1$. Celotno rekonstruirano WT vidimo na sliki 6.3.

Postopek rekonstrukcije zaporedja začnemo v skrajno levem listu drevesa. Iz trenutne abecede vidimo, da bit 0 ustreza presledku, bit 1 pa znaku a, ki je dvojček skrajno levemu listu. S to informacijo iz zaporedja bitov ⟨101⟩ rekonstruiramo v starševskem vozlišču zaporedje znakov ⟨a␣ a⟩ (glej sliko 6.4). Ker desna podveja levega poddrevesa še ni določena, se spustimo do najnižjega nivoja, kjer rekonstruiramo zaporedje z abecedo {i j} (zaporedje ⟨iijji⟩). S pomočjo simbola e, ki pride z leve veje, rekonstruiramo zaporedje, ki sestoji iz abecede {e i j}. Postopek je naslednji. Iz levega in desnega poddrevesa po vrsti vzamemo znake in jih zapisujemo v rekonstruirano zaporedje glede na vrednosti bitov v vozlišču. Iz desnega poddrevesa prihaja zaporedje, katerega črke po vrsti polagamo pod bite 1, iz levega poddrevesa dobimo znak e, ki ga postavimo pod bit 0. Po istem postopku rekonstruiramo celotno podzaporedje levega poddrevesa. Nato se spustimo na desno vejo drevesa in z enakim postopkom rekonstruiramo znake desnega podzaporedja. Obe podzaporedji nato uporabimo za dokončno rekonstrukcijo zaporedja, kot vidimo na sliki 6.4.



Slika 6.4: Rekonstruiranje zaporedja

Drevo valjčkov ima veliko možnih aplikacij. Obsežne preglede najdemo v [63, 64].

6.6 Burrows-Wheelerjeva transformacija

Ko razmišljamo o možnih transformacijah zaporedij, pomislimo tudi na permutacije. Nekatere permutacije bi zagotovo bile zelo ugodne za različne nadaljnje analize. Iz poljubnega zaporedja po določenem številu permutacij pridemo tudi do urejene permutacije, za katero pa vemo, da nam omogoča množico koristnih operacij (na primer iskanje elementa z bisekcijo ali stiskanje podatkov z RLE). Žal pa število permutacij prehitro narašča, da bi pristop bil uporaben za daljša zaporedja. Zaporedja sicer znamo urediti, a urejanje je, brez velikih količin dodatnih informacij, nepovraten postopek, kar nam pri stiskanju podatkov ne pomaga veliko.

Dobro nadomestilo urejenemu zaporedju pa nudi eden najbolj presenetljivih algoritmov, ki je bil kadarkoli izumljen v računalništvu, to je **Burrows-Wheelerjeva transformacija** (angl. Burrows-Wheeler transform, BWT) [65, 66]. BWT sprejme zaporedje $I = \langle \sigma_i \rangle$, $\sigma_i \in \Sigma$, in ga transformira v izhodno zaporedje $O = \langle \sigma_i \rangle$, $\sigma_i \in \Sigma$, $n = |I| = |O|$. BWT torej ne spremeni domene, prav tako ne dolžine obeh zaporedij, spremeni pa vrstni red simbolov in sicer tako, da pogosto združi enake simbole. Prav ta delna urejenost rezultata Burrows-Wheelerjeve transformacije pa omogoča množico njenih aplikacij (iskanje vzorcev v zaporedjih, iskanje najdaljšega skupnega podzaporedja, stiskanje podatkov) [66]. Pri tem je ključnega pomena, da obstaja tudi inverzna Burrows-Wheelerjeva transformacija – IBWT, ki za svoje delovanje potrebuje samo konstantno velik drobec informacije, to je tako imenovani **indeks BWT**.

Originalni postopek transformacije BWT, kot sta ga predlagala Burrows in Wheeler [65], si bomo ogledali s primerom 6.13.

$$I = \langle \texttt{regaregakvak} \rangle$$

Primer 6.13: Vhodno zaporedje za prikaz delovanja BWT

BWT najprej permutira I. Ker pa je vseh možnih permutacij preveč, tvori le n permutacij, ki jih pridobimo z n krožnimi premiki I (glej primer 6.14).

Zaporedja nato leksikografsko uredimo (primer 6.15). BWT predstavlja zaporedje simbolov v zadnjem stolpcu (v bistvu je to še ena od permutacij vhodnega zaporedja I). Da bi lahko rekonstruirali izvorno zaporedje, potrebujemo številko vrstice iz leksikografsko urejenih zaporedij, v kateri je I; to je indeks BWT, ki ga bomo označili kot i_{BWT}. V našem primeru je $i_{BWT} = 10$. Končen rezultat transformacije kaže primer 6.16. Ugotovimo, da so enaki znaki združeni (v tem primeru celo idealno združeni), kar je za

0	regaregakvak
1	egaregakvakr
2	garegakvakre
3	aregakvakreg
4	regakvakrega
5	egakvakregar
6	gakvakregare
7	akvakregareg
8	kvakregarega
9	vakregaregak
10	akregaregakv
11	kregaregakva

Primer 6.14: Prvi korak BWT

stiskanje podatkov odlična novica.

0	akregaregakv	
1	akvakregareg	
2	aregakvakreg	
3	egakvakregar	
4	egaregakvakr	
5	gakvakregare	
6	garegakvakre	
7	kregaregakva	
8	kvakregarega	
9	regakvakrega	
10	regaregakvak	←
11	vakregaregak	

Primer 6.15: Drugi korak BWT; simboli BWT so napisani z rdečo; puščica označuje položaj indeksa i_{BWT}

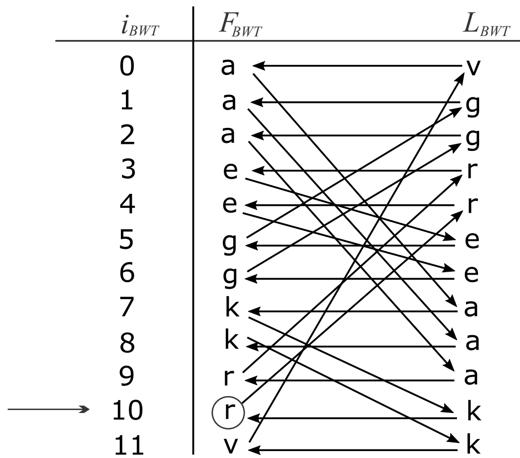
(a) $O = \langle$vggrreeaaakk$\rangle$

(b) i_{BWT}=10

Primer 6.16: (a) Transformirano zaporedje z BWT in (b) BWT-indeks

Inverzna Burrows-Wheelerjeva transformacija najprej zgradi dve zaporedji:

- zaporedje L_{BWT} hrani transformacijo samo in predstavlja zadnji stolpec, kot kaže primer 6.16;
- zaporedje F_{BWT}, ki ustreza prvemu stolpcu, dobimo z urejanjem stolpca L_{BWT}, kot kaže primer 6.16.



Slika 6.5: Postopek IBWT

Rekonstrukcijo začnemo z znakom na položaju i_{BWT} =10 v zaporedju F_{BWT}. Prvi rekonstruiran simbol je $F_{BWT_{10}} =$ r. V F_{BWT} sta dva simbola r, naleteli smo na drugega. Svojo pozornost zdaj prenesemo v zaporedje L_{BWT}, kjer poiščemo drugi simbol r. Najdemo ga na položaju 4. S premikom v zaporedje F_{BWT} na položaj 4 rekonstruiramo črko e. Postopek nadaljujemo (sledimo puščicam na sliki 6.5), dokler ne pridemo do položaja 10 v polju L_{BWT}, ko rekonstrukcijo zaključimo.

Največja slabost opisanega postopka za določitev transformacije BWT je njena časovna in prostorska zahtevnost. Leksikografsko urejanje n zaporedij dolžine n zahteva $O(n^2 log(n))$ časa, potrebujemo tudi $O(n^2)$ prostora. To je močno omejevalo uporabo BWT pri daljših zaporedjih, zato so takšna zaporedja razdelili v bloke (to sta predlagala tudi Burrows in Wheeler v svojem tehniškem poročilu [65]). A raziskovalci so odkrili, da je možno BWT sestaviti v linearnem času iz priponskega polja oziroma priponskega drevesa. Če bi znali sestaviti priponsko drevo ali priponsko polje v linearnem času, bi sestavili tudi BWT v linearnem času. Temu se bomo posvetili v naslednjem poglavju.

Naloge

1. Zaporedje $I = \langle \mathsf{004324321111} \rangle$ transformirajte s transformacijo premik naprej. Preverite, ali se je informacijska entropija spremenila. Opravite tudi inverzno transformacijo.
2. Implementirajte MTF in IMTF. Nato poiščite besedilo v slovenskem jeziku in ga transformirajte s transformacijo MTF. Preverite, ali se je informacijska entropija bistveno spremenila ter komentirajte rezultat.
3. Zajemite zaslonsko sliko besedila v slovenskem jeziku iz prejšnje naloge in ga shranite v formatu BMP. Preberite piksle slike (za branje formata BMP uporabite prostodostopne knjižnice ali sami implementirajte bralnik formata) in nad zaporedjem pikslov opravite transformacijo MTF. Tudi tokrat preverite, ali se je informacijska entropija transformiranega zaporedja spremenila.
4. Nalogi iz točk 2 in 3 realizirajte s transformacijami inverzne frekvence, kodiranje razdalj in desno manjše. Tudi za te transformacije izračunajte informacijsko entropijo.
5. Razmislite, kakšni morajo biti vhodni podatki, da je smiselno uporabiti transformacijo MTF večkrat zaporedoma. Pokažite s primerom.
6. Za zaporedje $I = \langle \textsf{siva kučma, bela brada} \rangle$ opravite transformacijo z drevesom valčkov. Zaporedje bitov nato uporabite za rekonstrukcijo izvornega zaporedja.
7. Z izvornim postopkom skonstruirajte Burrows-Wheelerjevo transformacijo za zaporedje $I = \langle \textsf{trla baba lan} \rangle$. Pravilnost rezultata preverite z inverzno transformacijo.
8. Implementirajte izvorni postopek Burrows-Wheelerjeve transformacije in ustrezno inverzno transformacijo. Z meritvami porabljenega časa CPU preverite časovno zahtevnost vaše implementacije.
9. Za podatke iz nalog 2 in 3 s svojim programom tvorite Burrows-Wheelerjevo transformacijo, za njo pa uporabite transformacijo premik naprej. Izračunajte informacijsko entropijo in jo primerjajte z rezultatom iz nalog 2 in 3.
10. Namesto transformacije premik naprej uporabite transformacijo inverzne frekvence za Burrows-Wheelerjevo transformacijo. Katera transformacija je uspešneje zmanjšala informacijsko entropijo?

Poglavje 7

Priponska polja in priponska drevesa

7.1 Priponsko polje

Priponsko polje (angl. suffix array) je urejeno zaporedje vseh pripon vhodnega zaporedja $I = \langle \sigma_i \rangle$, $0 \leq i < |I|$, $n = |I|$, $\sigma_i \in \Sigma$. j-ta pripona vsebuje vse $\sigma_i \in I$ od indeksa j do konca zaporedja, kar bomo označili kot $I_{j,n-1}$, $0 \leq j < n$. Najdaljša pripona $I_{0,n-1} = I$. Poleg tega, da je lahko vmesni korak za konstrukcijo transformacije BWT, ima priponsko polje številne druge aplikacije.

Priponsko polje sta uvedla Manber in Myers [67] in predstavila tudi algoritem za njegovo konstrukcijo. Razložimo ga s primerom 7.1a. Zaporedje vedno zaključimo s stražarjem $\sigma_s \neq \Sigma$, za katerega velja, da je, leksikografsko gledano, manjši od drugih znakov v abecedi, torej, $\sigma_s < \sigma_k$, $0 \leq k < |\Sigma|$. Tako je zaporedje, ki ga bomo uporabili za tvorbo priponskega polja, zaporedje $J = I + \sigma_s$, $m = |J|$ za stražarja pa bomo uporabili simbol $\sigma_s = !$, kot kaže primer 7.1b.

(a) $I = \langle \texttt{rabarbara} \rangle$
(b) $J = \langle \texttt{rabarbara!} \rangle$
(c) $m = 10$
(d) $\Sigma = \{\texttt{a b r}\}$

Primer 7.1: (a) Vhodno zaporedje, (b) zaporedje s stražarjem, (c) dolžina zaporedja J, (d) abeceda

Algoritem Mamber-Mayersa deluje v dveh korakih. V prvem tvori vse

pripone $J_{j,m}$, $0 \leq j < m$, kot kaže primer 7.2, v drugem pa pripone leksiko-

```
0 rabarbara!
1 abarbara!
2 barbara!
3 arbara!
4 rbara!
5 bara!
6 ara!
7 ra!
8 a!
9 !
```

Primer 7.2: Tvorba pripon $J_{j,m}$, kjer so indeksi pripon na levi strani

grafsko uredi (primer 7.3a). Indekse urejenih pripon zapišemo v zaporedje U, ki je iskano priponsko polje (primer 7.3b). U interpretiramo na naslednji način, na primer četrta pripona zaporedja J, urejena po abecedi, je $J_{6,10} = ara!$ oziroma $I_{6,9} = ara$.

```
9 !
8 a!
1 abarbara!
6 ara!
3 arbara!
5 bara!
2 barbara!
7 ra!
0 rabarbara!
4 rbara!
```

(a)

$U = \langle 9\ 8\ 1\ 6\ 3\ 5\ 2\ 7\ 0\ 4 \rangle$

(b)

Primer 7.3: (a) Urejene pripone in (b) priponsko polje

Poglejmo, kako iz priponskega polja U dobimo transformacijo BWT. Postopek razložimo s primerom 7.4. $U_0 = 9$ kaže na znak $J_9 = !$. Njegov predhodnik je simbol a, ki postane prvi znak zaporedja $O = \langle \mathsf{a} \rangle$. Postopek nadaljujemo: $U_1 = 8$, $J_8 = \mathsf{a}$, $J_{8-1} = \mathsf{r}$, zato $O = \langle \mathsf{ar} \rangle$. Poglejmo še primer, ko je $U_8 = 0$, $J_0 = r$, njegov predhodnik pa je $J_9 = !$. Rezultat

kaže primer 7.5a. Končno transformacijo BWT dobimo tako, da stražarja izločimo, na njegovo mesto pa premaknemo prvi simbol iz zaporedja O, označen z rdečo barvo na primeru 7.5b. Njegov položaj določa i_{BWT}. Kot vidimo, je tudi tokrat transformacija BWT skoraj popolnoma združila enake simbole in dobro opravila svoje delo. Bralec lahko za vajo opravi IBWT.

$$
\begin{aligned}
i:&\quad 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\\
U =&\ \langle \texttt{9 8 1 6 3 5 2 7 0 4} \rangle\\
J =&\ \langle \texttt{r a b a r b a r a !} \rangle
\end{aligned}
$$

Primer 7.4: Priponsko polje in vhodno zaporedje J

(a) $O = \langle \texttt{arrbbraa!a} \rangle$
(b) $O = \langle \texttt{rrbbraaaa} \rangle$ $\qquad i_{BWT} = 7$

Primer 7.5: Transformacija BWT, pridobljena iz priponskega polja U

Vidimo, da je iz priponskega polja možno sestaviti transformacijo BWT v linearnem času. Žal pa algoritem Mamber-Mayersa ni bistveno učinkovitejši od izvornega postopka Burrows-Wheelerja, saj ostajata časovna in prostorska zahtevnost nespremenjeni. Zato so se raziskovalci osredotočili na iskanje učinkovitejšega algoritma, ki bi znal sestaviti priponsko polje v boljši časovni zahtevnosti [68, 69]. Enega od njih si bomo ogledali v naslednjem podpoglavju.

7.1.1 Algoritem DC3

Algoritem DC3 za tvorbo priponskega polja je razvil Kärkkäinen s sodelavci [69]. Vhodni niz razdrobimo na krajša zaporedja, določimo njihove indekse v vhodnem nizu in le-te uredimo s korenskim urejanjem. Postopek rekurzivno ponavljamo, dokler obstajajo enaki trojčki v krajših zaporedjih. Končni rezultat je polje indeksov, ki predstavljajo priponsko polje.

Razlago algoritma bomo opravili s primerom, ki ga povzemamo po [70]. Vhodno zaporedje I in abecedo Σ kaže primer 7.6. Najprej tvorimo zaporedji indeksov B_1 in B_2 z enačbo 7.1, ki ju kažeta primera 7.7a in primera 7.7b. Zaporedji B_1 in B_2 nato zlepimo in dobimo zaporedje B_{12} (glej primer 7.7c).

$$
\begin{aligned}
B_1 &= \{i \in [0, n],\ \ i \pmod 3 = 1\}\\
B_2 &= \{i \in [0, n],\ \ i \pmod 3 = 2\}
\end{aligned}
\tag{7.1}
$$

$$\begin{array}{ll} & i: \quad 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11 \\ \texttt{(a)} & I = \langle \texttt{y a b b a d a b b a d o} \rangle \\ \\ \texttt{(b)} & \Sigma = \{\texttt{a b d o y}\} \end{array}$$

Primer 7.6: (a) Vhodno zaporedje in (b) abeceda za prikaz delovanja algoritma DC3

$$\begin{array}{ll} \texttt{(a)} & i: \quad 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12 \\ & I = \langle \texttt{y a b b a d a b b a d o} \rangle \\ \\ \texttt{(b)} & B_1 = \langle \texttt{1 4 7 10} \rangle \\ & B_2 = \langle \texttt{2 5 8 11} \rangle \\ \\ \texttt{(c)} & B_{12} = \langle \texttt{1 4 7 10 2 5 8 11} \rangle \end{array}$$

Primer 7.7: (a) Prikazani elementi zaporedij B_1 (zeleno) in B_2 (rdeče), (b) zaporedji B_1 in B_2 ter (c) zlepljeno zaporedje B_{12}

Nato tvorimo zaporedje D_1 tako, da sestavimo trojčke znakov, ki se začnejo na položajih, shranjenih v polju B_1 (primer 7.8a). Če trojčka ne moremo sestaviti, dodamo znak – stražarja !, za katerega velja $! \notin \Sigma$ in $! < \sigma_i$. Podobno tvorimo polje D_2 iz indeksov, shranjenih v B_2 (glej primer 7.8b). Polji D_1 in D_2 zlepimo v polje D (glej primer 7.8c). Shematično prikažemo rezultat še s primerom 7.9.

$$\begin{array}{ll} \texttt{(a)} & D_1 = \langle \texttt{|a b b|a d a|b b a|d o !|} \rangle \\ \texttt{(b)} & D_2 = \langle \texttt{|b b a|d a b|b a d|o ! !|} \rangle \\ \texttt{(c)} & D\ \ = \langle \texttt{|a b b|a d a|b b a|d o !|b b a|d a b|b a d|o ! !|} \rangle \end{array}$$

Primer 7.8: Zaporedji (a) D_1 in (b) D_2 hranita trojčke znakov, ki ju zlepimo v (c) skupno zaporedje D

$$\begin{array}{l} i: \quad 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13 \\ I\ = \langle \texttt{y a b b a d a b b a d o} \rangle \\ D_1 = \quad \langle \texttt{|a b b|a d a|b b a|d o !|} \rangle \\ D_2 = \qquad \langle \texttt{|b b a|d a b|b a d|o ! !|} \rangle \end{array}$$

Primer 7.9: Stanje po konstrukciji polj D_1 in D_2

Trojčke iz D leksikografsko uredimo s stabilnim algoritmom urejanja

(avtorji predlagajo korensko urejanje) in vsakemu trojčku priredimo nivo (angl. rank), ki ga zapišemo v zaporedje N. Nivo dobimo tako, da urejenim različnim trojčkom inkrementalno povečujemo vrednost njihovega nivoja, začenši pri 1, kot kaže razpredelnica 7.1. Vidimo, da je pri $B_{12,2}$ in $B_{12,7}$ nivo enak 4, saj je njuno zaporedje črk enako. Nastalo sovpadanje razrešimo, kot bomo videli pozneje, z rekurzivnim klicem algoritma DC3.

Razpredelnica 7.1: Določitev nivojev po korenskem urejanju

vhod	urejeni trojčki	N	$B_{12,i}$
|a b b|	|a b b|	1	1
|a d a|	|a d a|	2	4
|b b a|	|b a d|	3	8
|d o !|	|b b a|	4	2
|b b a|	|b b a|	4	7
|d a b|	|d a b|	5	5
|b a d|	|d o !|	6	10
|o ! !|	|o ! !|	7	11

Vhodno zaporedje I smo tako pretvorili v zaporedji B_{12} in N. Iz zaporedja N, na podlagi vrstnega reda v zaporedju B_{12}, sestavimo novo vhodno zaporedje I^1, ki mu na koncu dodamo še tri ničle. Te nam bodo prišle prav pri poznejšem korenskem urejanju. Nivo rekurzije bomo pri drugih zaporedjih označili s potenco ^{i}, $0 < i \leq Q$, kjer Q označuje globino rekurzije. Trenutno stanje kaže primer 7.10.

$$
\begin{array}{ll}
i: & \texttt{0 1 2 3 4 5 6 7 8 9 10} \\
B_{12} & = \langle \texttt{1 4 7 10 2 5 8 11} \rangle \\
N & = \langle \texttt{1 2 4 6 4 5 3 7} \rangle \\
I^1 & = \langle \texttt{1 2 4 6 4 5 3 7 0 0 0} \rangle
\end{array}
$$

Primer 7.10: Tvorba vmesnega zaporedja I^1, ki bo vhod v rekurzivni del algoritma DC3

Prvi rekurzivni klic algoritma DC3 nad zaporedjem I^1 poteka zelo podobno, kot smo opisali do zdaj. Najprej določimo vzorčne pripone in zaporedje D^1 (primer 7.11). Trojke polja D^1 nato uredimo (razpredelnica 7.2).

Ugotovimo, da so elementi zaporedja N^1 enolični, saj noben trojček nima enakega nivoja, zato rekurzijo zaključimo. Z enačbo 7.2 sestavimo še

$$
\begin{array}{lcl}
i: & & \texttt{0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17} \\
I^1 & = & \langle \texttt{1 2 4 6 4 5 3 7 0 0 0} \rangle \\
B^1_1 & = & \langle \texttt{1 4 7} \rangle \\
B^1_2 & = & \langle \texttt{2 5 8} \rangle \\
B^1_{12} & = & \langle \texttt{1 4 7 2 5 8} \rangle \\
D^1_1 & = & \langle \texttt{|2 4 6|4 5 3|7 0 0|} \rangle \\
D^1_2 & = & \langle \texttt{|4 6 4|5 3 7|0 0 0|} \rangle \\
D^1 & = & \langle \texttt{|2 4 6|4 5 3|7 0 0|4 6 4 |5 3 7 |0 0 0|} \rangle
\end{array}
$$

Primer 7.11: Stanje zaporedij v rekurzivnem klicu nivoja 1

Razpredelnica 7.2: Stanje po korenskem urejanju na prvem nivoju rekurzije

vhod	urejeni trojčki	N^1	$B^1_{12,i}$
|2 4 6|	|0 0 0|	1	8
|4 5 3|	|2 4 6|	2	1
|7 0 0|	|4 5 3|	3	4
|4 6 4|	|4 6 4|	4	2
|5 3 7|	|5 3 7|	5	5
|0 0 0|	|7 0 0|	6	7

zaporedje indeksov $B^1_0 = \langle 0\ 3\ 6 \rangle$.

$$
B^1_0 = \{i \in [0, n], \quad i \pmod 3) = 0\} \tag{7.2}
$$

Tvorimo pare znakov, ki jih bomo uredili. Pare znakov dobimo iz B^1_0 in iz nivojev urejenih trojčkov, kot kaže primer 7.12. Prvi znak v paru je znak, na katerega kaže indeks iz polja B^1_0. Pare uredimo s korenskim urejanjem, rezultat pa kaže razpredelnica 7.3.

$$
\begin{array}{lcl}
i: & & \texttt{0 1 2 3 4 5 6 7 8 9 10} \\
I_1 & = & \langle \texttt{1 2 4 6 4 5 3 7 0 0 0} \rangle \\
N^1 & = & \langle \texttt{? 2 4 ? 3 5 ? 6 1 0 0} \rangle
\end{array}
$$

Primer 7.12: Konstrukcija parov za urejanje na prvem nivoju rekurzije

V zadnjem koraku rekurzije zlijemo urejeni zaporedji B^1_0 in B^1_{12}, (glej primer 7.13). Predpostavimo, da v zaporedje B^1_0 kaže indeks i, v zaporedje B^1_{12} pa indeks j. Za zlivanje uporabimo pravili, ki ju izberemo z enačbo 7.3:

Pravilo P1. Najprej primerjamo znaka indeksov I_i in I_j. Če sta znaka enaka, primerjamo nivoja na položajih N_{i+1} in N_{j+1}.

Razpredelnica 7.3: Pari, ki jih uredimo najprej na prvem nivoju rekurzije

vhod	urejeni pari	B_0^1
|1 2|	|1 2|	0
|6 3|	|3 6|	6
|3 6|	|6 3|	3

$$
\begin{array}{ll}
i: & 0\ 1\ 2\ 3\ 4\ 5 \\
B_0^1 & = \langle 0\ \ 6\ \ 3 \rangle \\
B_{12}^1 & = \langle 8\ \ 1\ \ 4\ \ 2\ \ 5\ \ 7 \rangle
\end{array}
$$

Primer 7.13: Zaporedji B_0^1 in B_{12}^1

Pravilo P2. Primerjamo znaka na položajih indeksov $B_{0,i}$ in $B_{12,j}$. Če najdemo ujemanje, preverimo znaka na položajih $B_{0,i+1}$ in $B_{12,j+1}$. Če ponovno najdemo ujemanje, preverimo nivoja na N_{i+2} in N_{j+2}.

$$B_{12,j} \pmod 3 = \begin{cases} 1: & \text{izberi } P1 \\ \text{sicer:} & \text{izberi } P2 \end{cases} \tag{7.3}$$

Vse primerjave nikoli ne morejo biti uspešne, saj pred primerjanjem zagotovimo, da so vse vrednosti nivojev v polju N enolične. Poglejmo postopek zlivanja s primerom.

1. Na začetku sta $i = j = 0$, zato $B_{0,i=0}^1 = 0$ in $B_{12,j=0}^1 = 8$ (glej primer 7.13). Ker je ($8 \mod 3 = 2$), uporabimo pravilo P2. $B_{0,0}$ določa elemente $I_0^1 = 1$, $I_1^1 = 2$, in $N_2^1 = 4$, torej zaporedje $\langle 1\ 2\ 4 \rangle$. Simbole za primerjavo iz $B_{12}^1 = 8$ določimo na podoben način: $I_8^1 = 0$, $I_9^1 = 0$, in $N_{10}^1 = 0$, torej $\langle 0\ 0\ 0 \rangle$. Ker je $\langle 0\ 0\ 0 \rangle < \langle 1\ 2\ 4 \rangle$, je prvi element zlivanja $8 \in B_{12}^1$. V naslednjem koraku zato vzamemo naslednji element iz B_{12}^1, to je, da postavimo $j = 1$.

2. $i = 0$, $j = 1$, $B_{0,0}^1 = 0$ in $B_{12,1}^1 = 1$. Ker je ($1 \mod 3 = 1$), vzamemo pravilo P1, ki pravi, da preverjamo prvi znak iz I^1 in drugi znak iz N^1. Zaporedje za primerjanje, določeno z B_0^1, je $\langle 1\ 2 \rangle$, zaporedje, določeno z $B_{12,1}^1$ pa $\langle 2\ 4 \rangle$. Ker je $\langle 1\ 2 \rangle < \langle 2\ 4 \rangle$, pošljemo na izhod O^1 $0 \in B_{0,0}^1$ in se v naslednjem koraku premaknemo na naslednji znak iz $B_{0,1}^1$ tako, da povečamo/inkrementiramo i.

3. $i = 1$, $j = 1$, $B^1_{0,1} = 6$, $B^1_{12,1} = 1$. Vzamemo pravilo P1. Zaporedji za primerjanje $B^1_0 = \langle 3\ 6 \rangle$ in $B^1_{12,1} = \langle 2\ 4 \rangle$; $\langle 2\ 4 \rangle < \langle 3\ 6 \rangle$, zato pošljemo na izhod 1 iz B^1_{12} in inkrementiramo j.

4. $i = 1$, $j = 2$, $B^1_{0,1} = 6$, $B^1_{12,2} = 4$. Uporabimo pravilo P1. Zaporedji za primerjanje $B^1_0 = \langle 3\ 6 \rangle$ in $B^1_{12,2} = \langle 4\ 5 \rangle$; ker je $\langle 3\ 6 \rangle < \langle 4\ 5 \rangle$, pošljemo na izhod 6 iz B^1_0 in povečamo i.

5. $i = 2$, $j = 2$, $B^1_{0,2} = 3$, $B^1_{12,2} = 4$; ker je ($4 \mod 3 = 1$), vzamemo pravilo P1. Zaporedji za primerjanje $B^1_{0,2} = \langle 6\ 3 \rangle$ in $B^1_{12,2} = \langle 4\ 5 \rangle$; ker je $\langle 4\ 5 \rangle < \langle 6\ 3 \rangle$, pošljemo na izhod 4 iz B^1_{12} ter povečamo j.

6. $i = 2$, $j = 3$, $B^1_{0,2} = 3$, $B^1_{12,3} = 2$; ker je ($2 \mod 3 = 2$), uporabimo P2. Zaporedji za primerjanje $B^1_{0,2} = \langle 6\ 4\ 5 \rangle$ in $B^1_{12,3} = \langle 4\ 6\ 3 \rangle$; $\langle 4\ 6\ 3 \rangle < \langle 6\ 4\ 5 \rangle$, 2 vstavimo v O^1 ter povečamo j.

7. $i = 2$, $j = 4$, $B^1_{0,2} = 3$, $B^1_{12,4} = 5$; ($5 \mod 3 = 2$); uporabimo P2. Zaporedji za primerjanje $B^1_{0,2} = \langle 6\ 4\ 5 \rangle$ in $B^1_{12,4} = \langle 5\ 3\ 6 \rangle$; $\langle 5\ 3\ 6 \rangle < \langle 6\ 4\ 5 \rangle$, zato gre v O^1 5 iz B^1_{12}, nato inkrementiramo j.

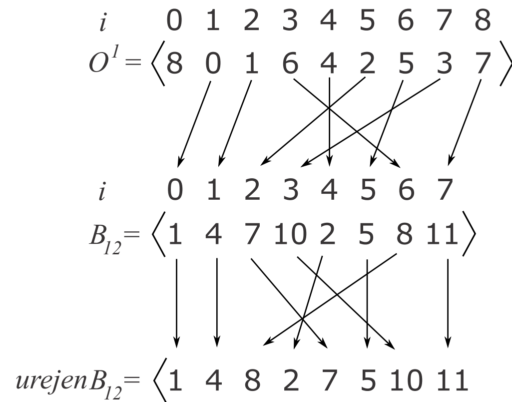
8. $i = 2$, $j = 5$, $B^1_{0,2} = 3$, $B^1_{12,5} = 7$; ($7 \mod 3 = 1$); uporabimo P1. Zaporedji za primerjanje $B^1_{0,2} = \langle 6\ 3 \rangle$ in $B^1_{12,5} = \langle 7\ 1 \rangle$; $\langle 6\ 3 \rangle < \langle 7\ 1 \rangle$; 3 iz B^1_0 dodamo v O^1, i pa povečamo za ena.

9. Ker smo uporabili vse znake iz enega zaporedja (B^1_0 v našem primeru), znake iz drugega zaporedja po vrstnem redu pošljemo na izhod (glej primer 7.14).

$$
\begin{array}{ll}
i & 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8 \\
O^1 & = \langle 8\ 0\ 1\ 6\ 4\ 2\ 5\ 3\ 7 \rangle
\end{array}
$$

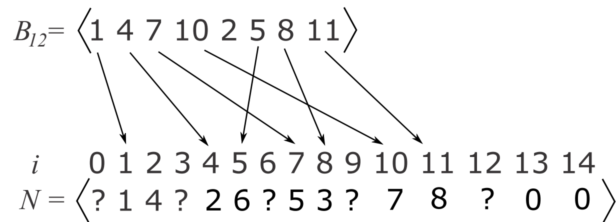
Primer 7.14: Rezultat zlivanja na prvem nivoju rekurzije

Rezultat O^1, ki smo ga dobili z zlivanjem in ga vrnemo iz rekurzije, uporabimo za urejanje B_{12}, kar storimo s sprehodom skozi zaporedje O^1, začenši pri indeksu 1. Prvega indeksa v polju O^1 z vrednostjo 8 namreč ne potrebujemo, ker kaže na dodan trojček $\langle 0\ 0\ 0 \rangle$. Vsako vrednost iz O^1 obravnavamo kot indeks v B_{12}. Postopek najlažje razložimo s sliko 7.1. Zaradi preglednosti smo označili, da rezultat shranimo v zaporedje $urejenB_{12}$, čeprav dejansko vrednosti v zaporedju B_{12} samo prepišemo.



Slika 7.1: Dokončno urejanje polja B_{12} po vrnitvi iz rekurzije

V zadnjem koraku moramo odpraviti še problem nivojev iz razpredelnice 7.1, saj je ta bil razlog rekurzivnega klica. Ta korak je enostaven. Sprehodimo se skozi urejeno polje B_{12}. Vsako vrednost uporabimo kot indeks v polje nivojev N, pri čemer števec nivojev inkrementalno povečujemo. Postopek prikazuje slika 7.2. Vidimo, da sovpadanja (dva nivoja z vrednostjo 4) ni več.



Slika 7.2: Določitev nivojev v zaporedju N po vrnitvi iz rekurzije

Ko imamo urejeno polje B_{12} in enolične indekse, lahko uredimo še neurejene pripone na enak način in z enakima praviloma, kot smo to opravili v rekurziji. Za vajo opravimo še to nalogo. Trenutno stanje kaže primer 7.15.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
I =	⟨y	a	b	b	a	d	a	b	b	a	d	o	!	!	!⟩
N =	⟨?	1	4	?	2	6	?	5	3	?	7	8	?	0	0⟩

Primer 7.15: Konstrukcija parov za končno urejanje

Z enačbo 7.2 določimo zaporedje indeksov $B_0 = \langle 0\ 3\ 6\ 9\ 12 \rangle$, tvorimo

pare $B_{0,i}$, N_{i+1} in jih uredimo s korenskim urejanjem, kot kaže razpredelnica 7.4.

Razpredelnica 7.4: Stanje po korenskem urejanju

vhod	urejeni pari	B_0
|y 1|	|! 0|	12
|b 2|	|a 5|	6
|a 5|	|a 7|	9
|a 7|	|b 2|	3
|! 0|	|y 1|	0

Zaporedji $B_0 = \langle \mathsf{12\ 6\ 9\ 3\ 0} \rangle$ in $B_{12} = \langle \mathsf{1\ 4\ 8\ 2\ 7\ 5\ 10\ 11} \rangle$ zlijemo in dobimo rezultat O. Poglejmo korake:

1. $i = j = 0$;
   $B_{0,i} = \mathsf{12}$, $B_{12,j} = \mathsf{1}$;
   $(1 \mod 3 = 1) \rightarrow \mathrm{P1}$;
   $I_{12} = \mathsf{!}$, $N_{13} = \mathsf{0}$;
   $I_1 = \mathsf{a}$, $N_2 = \mathsf{4}$;
   $\langle \mathsf{!\ 0} \rangle < \langle \mathsf{a\ 4} \rangle$;
   $O = \langle \mathsf{1\ 2} \rangle$.

2. $i = 0$, $j = 2$;
   $B_{0,i} = \mathsf{6}$, $B_{12,j} = \mathsf{1}$;
   $(1 \mod 3 = 1) \rightarrow \mathrm{P1}$;
   $I_6 = \mathsf{a}$, $N[7] = \mathsf{5}$;
   $I_1 = \mathsf{a}$, $N[2] = \mathsf{4}$;
   $\langle \mathsf{a\ 4} \rangle < \langle \mathsf{a\ 5} \rangle$;
   $O = \langle \mathsf{12\ 1} \rangle$.

3. $i = 0$, $j = 2$;
   $B_{0,i} = \mathsf{6}$, $B_{12,j} = \mathsf{4}$;
   $(4 \mod 3 = 1) \rightarrow \mathrm{P1}$;
   $I_6 = \mathsf{a}$, $N_7 = \mathsf{5}$;
   $I_4 = \mathsf{a}$, $N_5 = \mathsf{6}$;
   $\langle \mathsf{a\ 5} \rangle < \langle \mathsf{a\ 6} \rangle$;
   $O = \langle \mathsf{12\ 1\ 6} \rangle$.

4. $i = 1$, $j = 2$;
   $B_{0,i} = \mathsf{9}$, $B_{12_j} = \mathsf{4}$;

$(4 \mod 3 = 1) \rightarrow \text{P1};$
$I_9 = \mathsf{a},\ N_{10} = 7;$
$I_4 = \mathsf{a},\ N_5 = 6;$
$\langle \mathsf{a}\ 6 \rangle < \langle \mathsf{a}\ 7 \rangle;$
$O = \langle 12\ 1\ 6\ 4 \rangle.$

5. $i = 1,\ j = 3;$
   $B_{0,i} = 9,\ B_{12,j} = 8;$
   $(8 \mod 3 = 2) \rightarrow \text{P2};$
   $I_9 = \mathsf{a},\ I_{10} = \mathsf{d},\ N_{11} = 8;$
   $I_8 = \mathsf{b},\ I_9 = \mathsf{a},\ N_{10} = 7;$
   $\langle \mathsf{a}\ \mathsf{d}\ 8 \rangle < \langle \mathsf{b}\ \mathsf{a}\ 7 \rangle;$
   $O = \langle 12\ 1\ 6\ 4\ 9 \rangle.$

6. $i = 2,\ j = 3;$
   $B_{0,i} = 3,\ B_{12,j} = 8;$
   $(8 \mod 3 = 2) \rightarrow \text{P2};$
   $I_3 = \mathsf{b},\ I_4 = \mathsf{a},\ N_5 = 6;$
   $I_8 = \mathsf{b},\ I_9 = \mathsf{a},\ N_{10} = 7;$
   $\langle \mathsf{b}\ \mathsf{a}\ 6 \rangle < \langle \mathsf{b}\ \mathsf{a}\ 7 \rangle;$
   $O = \langle 12\ 1\ 6\ 4\ 9\ 3 \rangle.$

7. $i = j = 3;$
   $B_{0,i} = 0,\ B_{12_j} = 8;$
   $(8 \mod 3 = 2) \rightarrow \text{P2};$
   $I_0 = \mathsf{y},\ I_1 = \mathsf{a},\ N_2 = 4;$
   $I_8 = \mathsf{b},\ I_9 = \mathsf{a},\ N_{10} = 7;$
   $\langle \mathsf{b}\ \mathsf{a}\ 7 \rangle < \langle \mathsf{y}\ \mathsf{a}\ 4 \rangle;$
   $O = \langle 12\ 1\ 6\ 4\ 9\ 3\ 8 \rangle.$

8. $i = 3,\ j = 4;$
   $B_{0,i} = 0,\ B_{12,j} = 2;$
   $(2 \mod 3 = 2) \rightarrow \text{P2};$
   $I_0 = \mathsf{y},\ I_1 = \mathsf{a},\ N_2 = 4;$
   $I_2 = \mathsf{b},\ I_3 = \mathsf{b},\ N_4 = 2;$
   $\langle \mathsf{b}\ \mathsf{b}\ 2 \rangle < \langle \mathsf{y}\ \mathsf{a}\ 4 \rangle;$
   $O = \langle 12\ 1\ 6\ 4\ 9\ 3\ 8\ 2 \rangle.$

9. $i = 3,\ j = 5;$
   $B_{0,i} = 0,\ B_{12,j} = 7;$
   $(7 \mod 3 = 1) \rightarrow \text{P1};$
   $I_0 = \mathsf{y},\ N_1 = 1;$

$S_7 = \mathsf{b}$, $N_8 = 3$;
$\langle \mathsf{b}\ 3 \rangle < \langle \mathsf{y}\ 1 \rangle$;
$O = \langle 12\ 1\ 6\ 4\ 9\ 3\ 8\ 2\ 7 \rangle$.

10. $i = 3$, $j = 6$;
    $B_{0,i} = 0$, $B_{12,j} = 5$;
    $(5 \mod 3 = 2) \rightarrow \text{P2}$;
    $I_0 = \mathsf{y}$, $S_1 = \mathsf{a}$, $N_2 = 4$;
    $I_5 = \mathsf{d}$, $I_6 = \mathsf{a}$, $N_7 = 5$;
    $\langle \mathsf{d\ a}\ 5 \rangle < \langle \mathsf{y\ a}\ 4 \rangle$;
    $O = \langle 12\ 1\ 6\ 4\ 9\ 3\ 8\ 2\ 7\ 5 \rangle$.

11. $i = 3$, $j = 7$;
    $B_{0,i} = 0$, $B_{12,j} = 10$;
    $(10 \mod 3 = 1) \rightarrow \text{P1}$;
    $I_0 = \mathsf{y}$, $N_1 = 1$;
    $I_{10} = \mathsf{d}$, $N_{11} = 8$;
    $\langle \mathsf{d}\ 8 \rangle < \langle \mathsf{y}\ 1 \rangle$;
    $O = \langle 12\ 1\ 6\ 4\ 9\ 3\ 8\ 2\ 7\ 5\ 10 \rangle$.

12. $i = 3$, $j = 8$;
    $B_{0,i} = 0$, $B_{12,j} = 11$;
    $(11 \mod 3 = 2) \rightarrow \text{P2}$;
    $I_0 = \mathsf{y}$, $I_1 = \mathsf{a}$, $N_2 = 4$;
    $I_{11} = \mathsf{o}$, $S_{12} = !$, $N_{13} = 0$;
    $\langle \mathsf{o}\ !\ 0 \rangle < \langle \mathsf{y\ a}\ 1 \rangle$;
    $O = \langle 12\ 1\ 6\ 4\ 9\ 3\ 8\ 2\ 7\ 5\ 10\ 11 \rangle$.

13. Ker smo izčrpali vse elemente zaporedja B_{12}, na izhod pošljemo preostale elemente iz B_0. Končno priponsko polje vidimo kaže primer 7.16.

$$O = \langle 12\ \ 1\ \ 6\ \ 4\ \ 9\ \ 3\ \ 8\ \ 2\ \ 7\ \ 5\ \ 10\ \ 11\ \ 0 \rangle$$

Primer 7.16: Priponsko polje zaporedja $I = \langle \mathsf{yabbadabbado} \rangle$, povzeto po [70]

Vse urejene pripone kaže primer 7.17.

```
12: ⟨⟩
1: ⟨abbadabbado⟩
6: ⟨abbado⟩
4: ⟨adabbado⟩
9: ⟨ado⟩
3: ⟨badabbado⟩
8: ⟨bado⟩
2: ⟨bbadabbado⟩
7: ⟨bbado⟩
5: ⟨dabbado⟩
10: ⟨do⟩
11: ⟨o⟩
12: ⟨yabbadabbado⟩
```

Primer 7.17: Urejene pripone zaporedja $I = \langle \mathsf{yabbadabbado} \rangle$

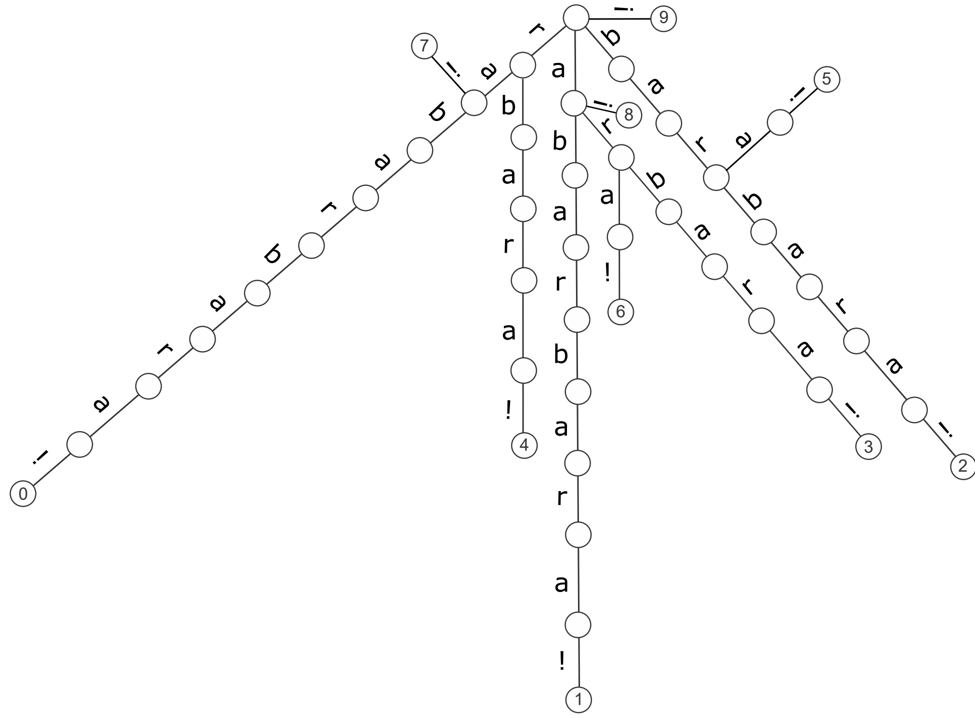
7.2 Številsko in priponsko drevo

Številsko drevo (angl. digital tree, prefix tree) je zelo stara podatkovna struktura [71, 72], namenjena predstavitvi množice zaporedij. Poznamo jo tudi pod imenom **iskalno drevo** (angl. re*trie*val tree) ali krajše **trie**. Trie pogosto uporabljamo za implementacijo slovarjev, saj na vhodu sprejme množico fraz, ki pa so lahko, tako kot bodo v našem primeru, tudi pripone zaporedja I.

Naj bo zaporedje $I = \langle \sigma_i \rangle$, $\sigma_i \in \Sigma$, dolžine $n = |I|$. Tudi tokrat zaporedje I razširimo s stražarjem $\sigma_s =!$, $\sigma_s \neq \Sigma$, tako da dobimo zaporedje J z dolžino $m = n + 1$. Številsko drevo $\mathcal{T}_D$ je drevo z m listi, kjer j-ti list ustreza j-ti priponi $J_{j,m}$. Vsaka povezava v $\mathcal{T}_D$ predstavlja pripono iz J. Nobena povezava iz danega vozlišča nima istega prvega znaka σ_i.

Oglejmo si situacijo za zaporedje J iz primera 7.1, iz katerega smo dobili pripone, ki jih kaže primer 7.2. Dobljeno številsko drevo vidimo na sliki 7.3.

Največja slabost številskega drevesa je njegova prostorska zahtevnost. Na sliki 7.4a vidimo $\mathcal{T}_D$ za najugodnejši primer, ko je $J = \langle \mathsf{xxxx!} \rangle$, $n = |J| - 1 = 4$. $\mathcal{T}_D$ ima n notranjih vozlišče in koren, skupno torej $n+1$ vozlišč. Poleg tega imamo še $n+1$ povezav s stražarjem $\sigma_s = !$, ki vodijo do listov. Skupaj ima $\mathcal{T}_D$ $2(n+1) = 10$ vozlišč, kar zahteva $O(n)$ prostora. V primeru na sliki 7.4b vidimo $\mathcal{T}_D$ za zaporedje $J = \langle \mathsf{xxxyyy!} \rangle$, kjer je $n = |J| - 1 = 6$. V y-povezavi imamo n vozlišč ($\frac{n}{2}$ notranjih in $\frac{n}{2}$ listov). V x-povezavi iz vsakega od $\frac{n}{2}$ x-vozlišč izhaja $\frac{n}{2}$ $y-$vozlišč in $\frac{n}{2}$ listov, kar nam skupaj za x-povezavo da $2\frac{n}{2} + (\frac{n}{2})^2$ vozlišč. Skupaj imamo torej $n + n + (\frac{n}{2})^2 + 2 =$



Slika 7.3: Številsko drevo zaporedja ⟨rabarbara!⟩

$2(n+1) + \frac{n^2}{4} = O(n^2)$ vozlišč ob upoštevanju korena in lista, ki izhaja iz njega, kar vsekakor ni dobra novica za daljša zaporedja.

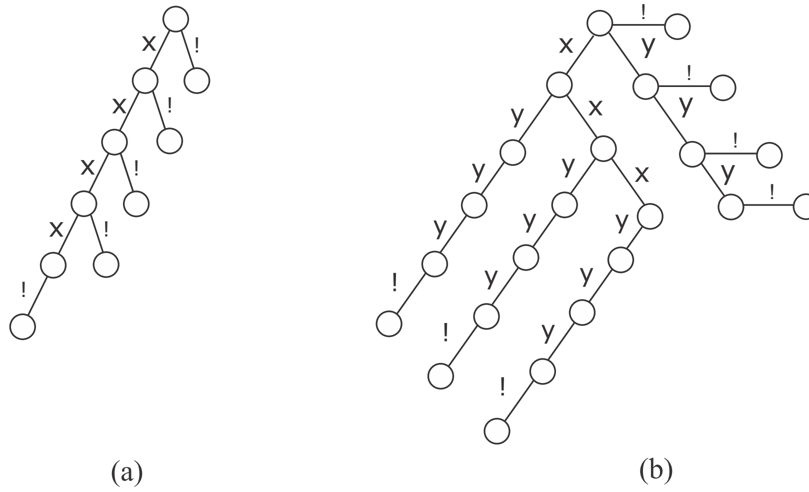
Prostorsko učinkovitejša predstavitev številskega drevesa je **priponsko drevo** $\mathcal{T}_P$. Priponsko drevo dobimo tako, da odstranimo vsa notranja vozlišča, ki imajo samo enega potomca, znake med takšnimi vozlišči pa zlepimo v zaporedja, kot je na sliki 7.5. Priponsko drevo $\mathcal{T}_P$, zgrajeno za zaporedje J, ima naslednje lastnosti:

1. Drevo ima $m = n + 1$ listov.

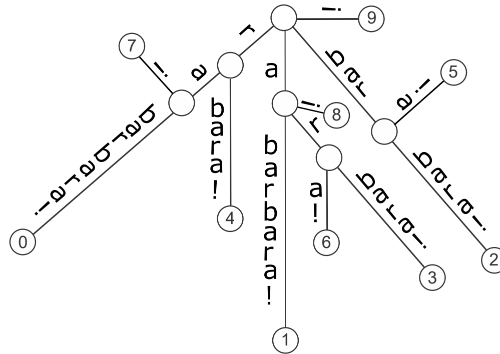
2. V drevesu je največ n notranjih vozlišč, kjer korensko vozlišče obravnavamo kot notranje.

3. Vsako notranje vozlišče ima vsaj 2 sinova.

Priponsko drevo ima torej toliko listov, kot je pripon, torej $n+1$. Koliko pa imamo notranjih vozlišč? Za vsako notranje vozlišče velja, da ima vsaj dva potomca. Če bi imelo vsako vozlišče samo dva potomca, bi dobili polno

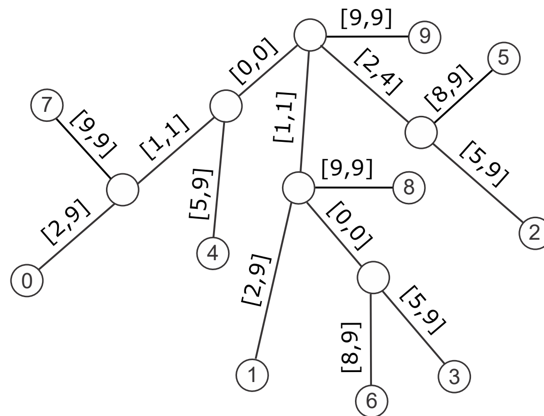


Slika 7.4: Število vozlišč v $\mathcal{T}_D$



Slika 7.5: Priponsko drevo je *stisnjeno* številsko drevo

binarno drevo, za katero vemo, da ima $n-1$ notranjih vozlišč. Ker ima $\mathcal{T}_P$ največ $2n+1$ vozlišč in največ n povezav med vozlišči, je njegova prostorska zahtevnost $O(n)$. Nadaljnje varčevanje s prostorom dosežemo z zamenjavo eksplicitnih oznak povezav z njihovimi kazalci v zaporedje J. S tem je število bitov, potrebnih za predstavitev zaporedja na povezavi, konstantno. V primeru več enakih oznak na povezavah najpogosteje uporabimo kar prvo pojavitev. Tako sestavljeno priponsko drevo za zaporedje J iz primera 7.1 vidimo na sliki 7.6.



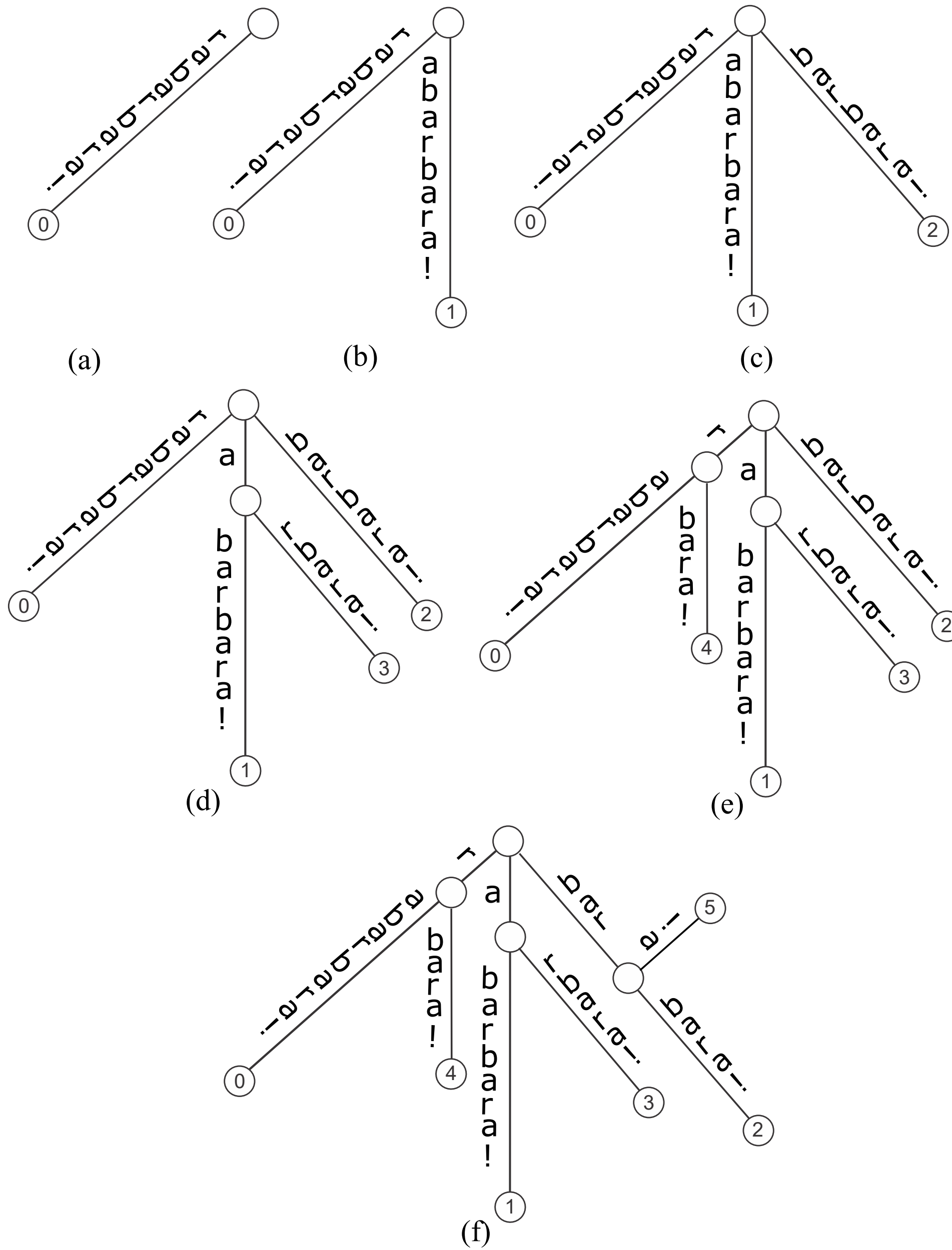
Slika 7.6: Priponsko drevo s kazalci v zaporedje $J =$⟨rabarbara!⟩

7.2.1 Konstrukcija priponskega drevesa z naivno metodo

Konstrukcija $\mathcal{T}_P$ je enostavna, če nas ne skrbi časovna zahtevnost. Nekaj korakov algoritma za pripone iz primera 7.2 vidimo na sliki 7.7, kjer smo, zaradi lažjega spremljanja postopka, na povezave drevesa namesto kazalcev vpisali podzaporedja.

Ko ustvarimo koren, vstavimo prvo pripono $J_{0,9}$ = ⟨rabarbara!⟩ kot prvo povezavo v $\mathcal{T}_P$ (slika 7.7a). Za vsako naslednjo pripono se od korena spuščamo po ustrezni povezavi, dokler obstaja ujemanje znakov. Za pripono $J_{1,9}$ = ⟨abarbara!⟩ ujemanja ne najdemo, zato povezavo priključimo h korenu, kot kaže slika 7.7b. Na isti način vstavimo tudi pripono $J_{2,9}$ = ⟨barbara!⟩ (slika 7.7c). Pri priponi $J_{3,9}$ =⟨arbara!⟩ ugotovimo, da iz korena že izhaja povezava, ki nosi pripono $J_{1,9}$, katere prvi znak je a. Zato se spustimo v to povezavo. A že drugi znak povzroči neujemanje. Pripono $J_{1,9}$ zato na mestu neujemanja razdelimo tako, da v povezavo vrinemo notranje vozlišče. Iz tega vozlišča zdaj izhajata dve povezavi; ena povezava hrani preostanek pripone $I_{1,9}$, druga pa preostanek pripone $J_{3,9}$ (slika 7.7d). Naslednja pripona je $J_{4,9}$ = ⟨rbara!⟩. Podobno kot pri prejšnjem primeru tudi tokrat povezava s prvo črko pripone v $\mathcal{T}_P$ že obstaja, priponi $J_{0,9}$ in $J_{4,9}$ pa se ujemata v dolžini enega znaka. Stanje kaže slika 7.7e. Oglejmo si še pripono $J_{5,9}$ = ⟨bara!⟩. Tokrat se priponi $J_{2,9}$ in $J_{5,9}$ ujemata v prvih treh znakih, zato vrinemo notranje vozlišče pred četrtim znakom, rezultat pa vidimo na sliki 7.7f. Postopek nadaljujemo, dokler ne obdelamo vseh pripon. Časovna zahtevnost naivnega algoritma je $O(n^2)$.

Za konstrukcijo priponskih dreves obstaja več učinkovitejših algoritmov,



Slika 7.7: Nekaj korakov konstrukcije priponskega drevesa z naivno metodo

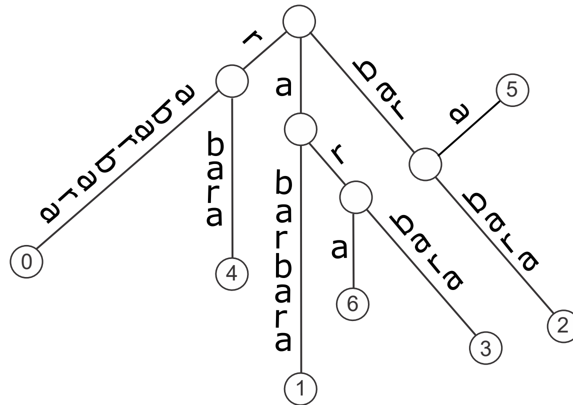
celo takšnih, ki delujejo v linearnem času. Prvega med njimi je razvil Weiner [73], najbolj znana pa sta McCreightov [74] in Ukkonenov algoritem [75].

Podrobneje si bomo ogledali slednjega.

7.2.2 Ukkonenov algoritem

Ukkonenov algoritem sestavi priponsko drevo $\mathcal{T}_P$ s sprehodom skozi zaporedje J od leve proti desni, s čimer omogoča tudi pretočno izvedbo. Začetno drevo $\mathcal{T}_P$ sestoji samo iz korena in ustreza praznemu zaporedju.

Ukkonenov algoritem lahko med gradnjo hrani tako imenovano **nepopolno priponsko drevo** (angl. implicit suffix tree). Dobili bi ga iz priponskega drevesa tako, da bi odstranili stražarja iz vseh označb na povezavah drevesa, odstranili vse povezave brez označbe in vsa vozlišča, ki imajo manj kot dva sinova. Primer nepopolnega priponskega drevesa kaže slika 7.8. Ugotovimo, da se priponi ⟨ra⟩ in ⟨a⟩ ne zaključita v listu drevesa. Iz nepopolnega priponskega drevesa dobimo **priponsko drevo** z vstavitvijo simbola, ki še ne obstaja v $\mathcal{T}_P$ (da to vedno zagotovimo, potrebujemo stražarja σ_s). Nepopolno priponsko drevo sicer hrani vse pripone danega zaporedja. Ni pa nujno, da vse pripone pristanejo v listih drevesa, oziroma, da so v drevesu eksplicitno shranjene.



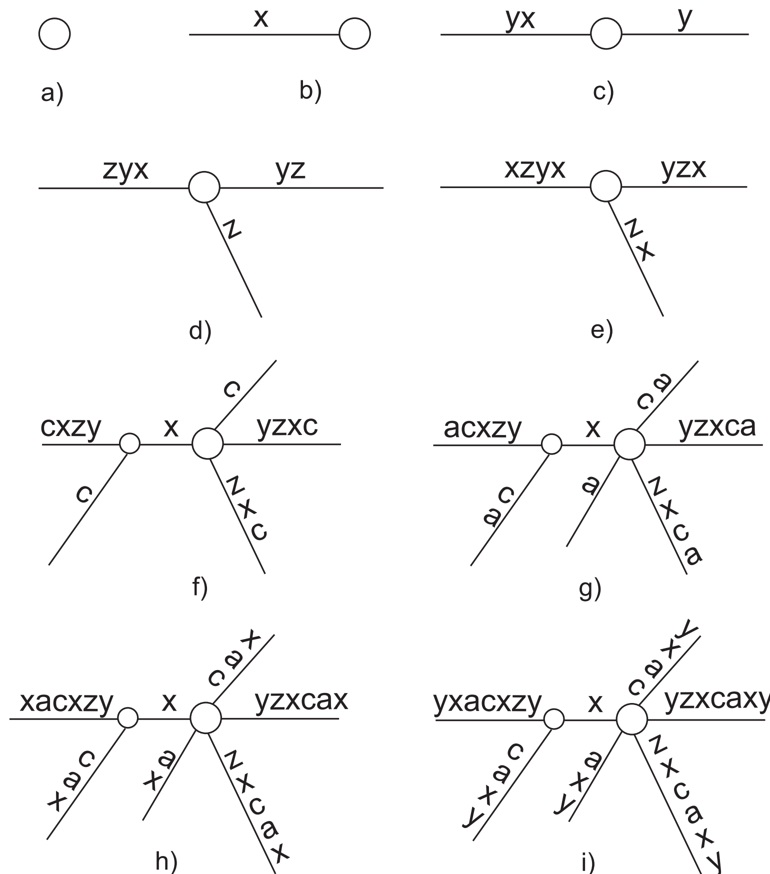
Slika 7.8: Nepopolno priponsko drevo za zaporedje $I = $ ⟨rabarbara⟩

Ukkonenovov algoritem sestavi $\mathcal{T}_P$ s tremi pravili glede na to, ali iz opazovanega vozlišča $a \in \mathcal{T}_P$ izhaja povezava s simbolom σ_i.

- **Pravilo P1**. Če iz vozlišča a ne izhaja nobena povezava, ki bi se začela s simbolom σ_i, dodamo novo povezavo z oznako σ_i.

- **Pravilo P2**. Iz vozlišča a že izhaja povezava s simbolom σ_i, povezava pa se zaključi v listu. Drevo $\mathcal{T}_P$ ažuriramo na ta način, da oznaki na povezavi prilepimo σ_i.

- **Pravilo P3**. Iz vozlišča a izhaja povezava, ki se začne z znakom σ_i, in se ne zaključi v listu. Algoritem v tem koraku ne naredi nič, $\mathcal{T}_P$ pa postane nepopolno priponsko drevo.



Slika 7.9: Razlaga Ukkonenovega algoritma s primerom zaporedja $I = \langle \mathsf{xyzxcaxy} \rangle$

Idejo Ukkonenovega algoritma bomo razložili s sliko 7.9, razlago pa povzemamo po [76]. Naj bo Γ generator znakov σ_i. Znak σ_i najprej dodamo v zaporedje I. Imejmo še drevo $\mathcal{T}_P$, ki na začetku sestoji samo iz korena (slika 7.9a). Prvi znak σ_0 iz Γ naj bo x, ki ga vstavimo v $I =\langle \mathsf{x} \rangle$. Uvedemo še indeksa i in j, ki kažeta v I. Po prihodu prvega elementa, kažeta nanj oba, torej $i = j = 0$ (slika 7.10a). Preverimo, ali je v $\mathcal{T}_P$ že povezava, ki se začne s $\sigma_0 =\langle \mathsf{x} \rangle$. Ker je ni, uporabimo pravilo P1. Ustvarimo novo povezavo z oznako x (slika 7.9b). Naj bo naslednji simbol $\sigma_1 = \mathsf{y}$, tako da je $I = \langle \mathsf{xy} \rangle$.

i postavimo na čelo zaporedja I, j pa na začelje, torej $i = 1$, $j = 0$, kot vidimo na sliki 7.10b. V drevesu preverimo, ali obstaja pripona $\langle \mathsf{xy} \rangle$. Povezava, ki se začne z x, obstaja, pripone $\langle \mathsf{xy} \rangle$ pa v drevesu še ni. Zato na povezavi x razširimo oznako z y po pravilu P2. Nato inkrementiramo indeks j, tako da sta $i = j = 1$. Pripona y v $\mathcal{T}_P$ še ne obstaja, zato jo vstavimo po pravilu P1. Rezultat vidimo na sliki 7.9c. Naslednji znak iz Γ naj bo $\sigma_2 = \mathsf{z}$, $I = \langle \mathsf{xyz} \rangle$, $j = 0$, $i = 2$ (slika 7.10c). Koraki algoritma so podobni kot prej. Ko je $j = 0$, preverimo, ali je v drevesu pripona $\langle \mathsf{xyz} \rangle$. Priponi $\langle \mathsf{xy} \rangle$ s pravilom P2 prilepimo z, nato povečamo j in preko povezave z oznako y s pravilom P2 tvorimo še pripono yz. Inkrementiramo j in s pravilom P1 vstavimo novo povezavo z oznako z. Rezultat vidimo na sliki 7.9d. Naslednji znak iz Γ naj bo ponovno x, postavimo $i = 3$ in $j = 0$ (slika 7.10d). V obstoječem $\mathcal{T}_P$ razširimo oznake obstoječih pripon tako, kot smo do zdaj s pravilom P2. Pri zadnjem znaku x preverimo, ali obstaja povezava, katerega oznaka se začne z x. Takšna povezava seveda obstaja. Zato se ustavimo in po pravilu P3 v tej povezavi ne naredimo ničesar. V preostalih dveh povezavah po pravilu P2 dodamo x (slika 7.9e). Drevo, ki smo ga dobili, ni popolno, saj se vse pripone ne zaključijo v listih. Tako se pripona $\langle \mathsf{x} \rangle$ se ne zaključi v listu. Kot smo povedali že prej, bo znak, ki še ne obstaja v $\mathcal{T}_P$, le-to spremenil v pravo priponsko drevo.

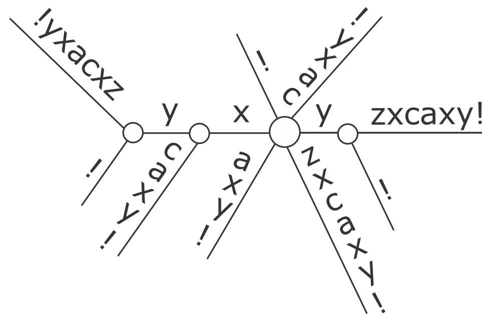
$$
\begin{array}{llllll}
I = & \langle \mathtt{x} \rangle & \langle \mathtt{xy} \rangle & \langle \mathtt{xyz} \rangle & \langle \mathtt{xyzx} \rangle & \langle \mathtt{xyzxc} \rangle \\
i: & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\
j: & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\
 & \mathtt{(a)} & \mathtt{(b)} & \mathtt{(c)} & \mathtt{(d)} & \mathtt{(e)}
\end{array}
$$

Slika 7.10: Začetne postaviteve indeksov i in j

Naj bo naslednji simbol iz Γ c. Indeksa i in j postavimo, kot kaže slika 7.10e. Za vrednosti $j = 0, 1, 2$ uporabimo pravilo P2 in obstoječe pripone razširimo z znakom c. Ko je $j = 3$, preverimo, ali iz korena izhaja kakšna povezava začenši s simbolom x. Takšna povezava seveda obstaja, to je pripona $I_{0,4} = \langle \mathsf{xyzxc} \rangle$. Ker pa naslednji znak v tej povezavi ni c, tvorimo notranje vozlišče; ena povezava vsebuje zaporedje $\langle \mathsf{yzxc} \rangle$, druga pa $\langle \mathsf{c} \rangle$. Inkrementiramo j ter obdelamo še zadnjo pripono, to je $\langle \mathsf{c} \rangle$. S pravilom P1 dodamo novo povezavo in dobimo situacijo, ki jo kaže slika 7.9f. Drevo, ki smo ga dobili, je priponsko drevo, saj se vse pripone zaporedja I zaključijo v listih drevesa.

V naslednjem koraku iz Γ dobimo simbol $\sigma_5 = \mathsf{a}$, $I = \langle \mathsf{xyzxca} \rangle$. S pravilom P2 na koncu vsake povezave prilepimo a. Ker iz korena $\mathcal{T}_P$ ne izhaja nobena povezava s prvim simbolom a, dodamo novo povezavo (slika 7.9g).

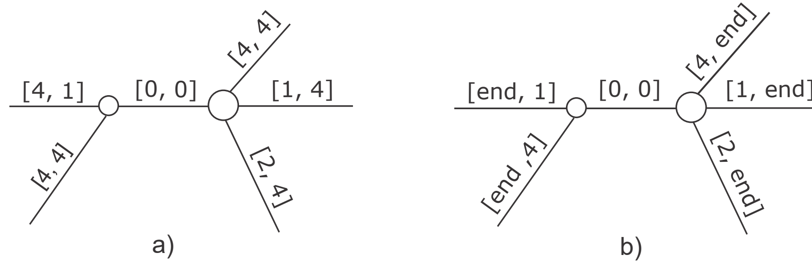
Naslednji simbol iz Γ naj bo x, torej je $I = \langle \mathsf{xyzxcax} \rangle$. S pravilom P2 prilepimo vsem povezavam, ki vodijo do listov, simbol x. Ker iz korena $\mathcal{T}_P$ že izhaja x, po pravilu P3 v zadnjem koraku ne naredimo ničesar. Rezultat kaže slika 7.9h. Zatem prispe simbol y. S pravilom P2 dodamo y na koncu vseh povezav, ki se zaključijo v listih $\mathcal{T}_P$. Ko pridemo z indeksom j do pripone $\langle \mathsf{xy} \rangle$, ugotovimo, da je pripona že v $\mathcal{T}_P$, zato ne naredimo ničesar. Nato Γ izda zadnji znak, stražarja $\sigma_s =!$, $J = \langle \mathsf{xyzxcaxy!} \rangle$. V vse povezave $\mathcal{T}_P$, ki hranijo liste pripon, prilepimo stražarja s pravilom P2. Ko obdelujemo pripono $\langle \mathsf{xy!} \rangle$, ugotovimo le delno ujemanje, zato s pravilom P1 vrinemo novo vozlišče v povezavi, ki se začne z x in nadaljuje z y. Podobno je, ko se premaknemo na pripono $\langle \mathsf{y!} \rangle$. Povezava z označbo, ki se začne z y, je že v $\mathcal{T}_P$, pripona $\langle \mathsf{y!} \rangle$ pa v $\mathcal{T}_P$ še ne obstaja, zato tudi tokrat s pravilom P1 vrinemo oglišče v povezavo. Nazadnje s pravilom P1 dodamo še povezavo s pripono $\langle ! \rangle$. Končno priponsko drevo vidimo na sliki 7.11. Čeprav ta trenutek še ni pomembno, opozorimo, da se v $\mathcal{T}_P$ nekatere oznake na povezavah ponavljajo (imamo kar tri povezave, na katerih je oznaka $\langle \mathsf{zxcaxy!} \rangle$).



Slika 7.11: Ob prihodu stražarja postane nepopolno priponsko drevo priponsko

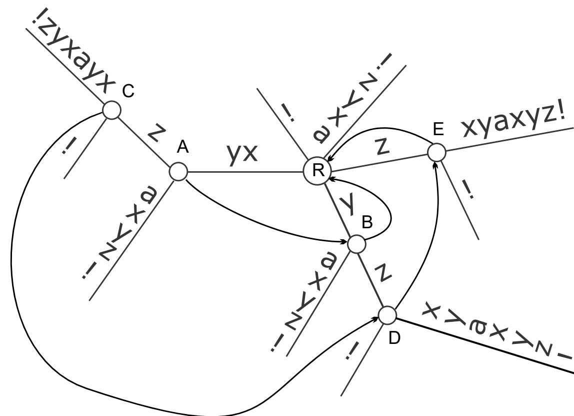
Analizo časovne zahtevnosti predstavljenega algoritma opravimo enostavno. V vsakem koraku se sprehodimo po vsaki priponi in s pravilom P2 ažuriramo vse dosedanje pripone. Pri tem obiščemo največ $\frac{1}{2}n(n+1)$ znakov, kar nam da časovno zahtevnost tega koraka $O(n^2)$. Ker imamo n korakov, je skupna časovna zahtevnost opisanega algoritma $O(n^3)$. Glavni krivec za tako visoko časovno zahtevnost je pravilo P2, saj smo v vsaki iteraciji morali obiskati vse povezave, ki so vodile do listov, in njihovim oznakam prilepiti nov simbol. Zato moramo pozornost prav gotovo najprej posvetiti temu koraku. Prvi korak v pravo smer je zamenjava označb povezav z indeksi v zaporedje I (oziroma J), kot smo to že spoznali (slika 7.6); postopku smo rekli **stiskanje oznak povezav**. Kot smo videli, se kazalec j po vsaki

iteraciji algoritma inkrementira. Zato lahko stopimo korak dlje in kazalec j nadomestimo s spremenljivko *end* (glej sliko 7.12). Po prihodu novega znaka pravilo P2 na ta način realiziramo v konstantnem času $O(1)$. Omenjen trik učinkovito rešuje povezave, ki vodijo v liste, za notranje povezave $\mathcal{T}_P$ pa poskrbimo z uvedbo **priponskih kazalcev**.



Slika 7.12: Realizacija pravila P2 v času $O(1)$

V $\mathcal{T}_P$ naj bo notranje vozlišče a ter povezava, ki izhaja iz njega z označbo $L(a) = \sigma_i\alpha$, kjer je σ_i znak ($\sigma_i \in \Sigma$), $\alpha \in \Sigma$ pa zaporedje, ki je lahko tudi prazno. Potem obstaja vozlišče b, $b \in \mathcal{T}_P$, ki ima oznako $L(b) = \alpha$. Povezavo med vozliščem a in b opravimo s priponskim kazalcem. Če je α prazno zaporedje, kazalec kaže na koren drevesa.



Slika 7.13: Priponski kazalci

Opravimo razlago s $\mathcal{T}_P$ na sliki 7.13. Pot do vozlišča z imenom A je pripona $\langle \mathsf{xy} \rangle$, kjer glede na zgornjo razlago velja, da je $\sigma_i = \mathsf{x}$ in $\alpha = \langle \mathsf{y} \rangle$. V priponskem drevesu obstaja vozlišče z imenom B, ki hrani pripono $\alpha = \langle \mathsf{y} \rangle$. Iz vozlišča A zato postavimo priponski kazalec na vozlišče B. V vozlišče B

smo prišli s pripono $\langle \mathsf{y} \rangle$, kjer je $\sigma_i = \mathsf{y}$ in $\langle\rangle$. Ker je α prazno zaporedje, postavimo priponski kazalec iz B na koren $\mathcal{T}_P$. Poglejmo vozlišče C. Pot do vozlišča C iz korena R je $\langle \mathsf{xyz} \rangle$, kjer je $\sigma_i = \mathsf{x}$, $\alpha = \langle \mathsf{yx} \rangle$. Pripona $\langle \mathsf{yz} \rangle$ se zaključi v vozlišču D, zato vozlišče C povežemo z vozliščem D s priponskim kazalcem. Do vozlišča D smo prišli s pripono $\langle \mathsf{yz} \rangle$, kjer je $\sigma_i = \mathsf{y}$ in $\alpha = \langle \mathsf{z} \rangle$. S pripono $\langle \mathsf{z} \rangle$ dosežemo tudi vozlišče E, zato nanj kaže priponski kazalec iz vozlišča D. Pripona $\langle \mathsf{z} \rangle$ sestoji iz $\sigma_i = \mathsf{z}$ in $\alpha = \langle\rangle$, zato kaže priponski kazalec iz E na koren drevesa.

Vsako notranje vozlišče ima priponski kazalec do drugega notranjega vozlišča ali do korena. Priponski kazalci omogočajo učinkovito pomikanje po drevesu ter vstavljanje notranjih vozlišč brez iskanja, s čimer zagotavljajo linearno časovno zahtevnost celotnega algoritma. Priponske kazalce vzpostavimo med konstrukcijo drevesa. Ko ustvarimo novo notranje vozlišče, nato pa v isti iteraciji še eno notranje vozlišče, ju povežemo s priponskim kazalcem. Zadnje ustvarjeno notranje vozlišče kaže na koren drevesa $\mathcal{T}_P$. Če ustvarimo samo eno notranje vozlišče, njegov priponski kazalec takoj postavimo na koren drevesa.

7.2.3 Realizacija Ukkonenovega algoritma

V nadaljevanju bomo ob primeru 7.18 pokazali, kako sestavimo $\mathcal{T}_P$ po Ukkonenovem algoritmu. Razlago povzemamo po [76].

$$\begin{array}{rl} i: & 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9 \\ J = & \langle \mathtt{x\ y\ z\ x\ y\ a\ x\ y\ z\ !} \rangle \end{array}$$

Primer 7.18: Vhodno zaporedje J za konstrukcijo priponskega drevesa z Ukkonenovim algoritmom

Algoritem uporablja naslednje spremenljivke – imenujemo jih tudi **aktivna točka** (angl. active point)):

- remaining (slov. preostalo) je spremenljivka, ki ima vrednost 0, če je $\mathcal{T}_P$ priponsko drevo, sicer hrani število pripon, ki se v drevesu ne zaključijo v listih;
- activeNode (slov. aktivno vozlišče) je vozlišče, na katerega smo naleteli med preiskovanjem pripone;
- activeEdge (slov. aktivna povezava) je kazalec v vhodno zaporedje J;
- activeLength (slov. aktivna dolžina) določa dolžino ujemanja pripone;

- end (slov. konec) je kazalec na zadnji znak trenutne pripone.

Algoritem in stanje v spremenljivkah razložimo v nadaljevanju po korakih:

Inicializacija. V inicializaciji ustvarimo koren drevesa ter inicializiramo spremenljivke (glej primer 7.19), ki bodo nadzirale izvajanje programa.

```
remaining = 0
activeNode = root
activeEdge = -1
activeLenght = 0
end = -1
```

Primer 7.19: Stanje aktivne točke – inicializacija

Prva iteracija ($i = 0$)**.** Najprej inkrementiramo remaining in end (primer 7.20a). Preverimo, ali je v drevesu že kakšna povezava, ki se začne z $J_0 = x$. Ker je ni, jo vstavimo v $\mathcal{T}_P$ (slika 7.14a), dekrementiramo remaining (glej primer 7.20b) in zaključimo prvo iteracijo.

```
remaining = 1          remaining = 0
activeNode = root      activeNode = root
activeEdge = -1        activeEdge = -1
activeLenght = 0       activeLenght = 0
end = 0                end = 0
     (a)                    (b)
```

Primer 7.20: Stanje aktivne točke – prva iteracija

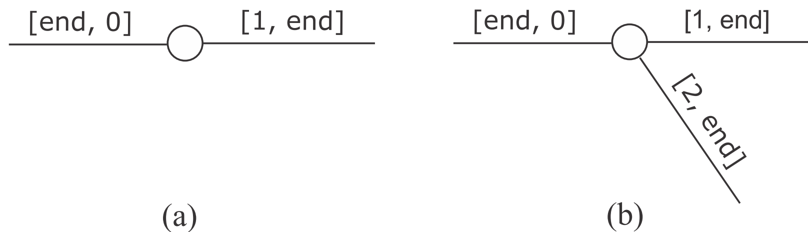
Druga iteracija ($i = 1$)**.** Ponovno najprej inkrementiramo remaining in end (primer 7.21a). Preverimo vrednost activeEdge. Ker je -1, tvorimo novo povezavo s pripono $\langle \mathsf{y} \rangle$. Dosedanjo situacijo kaže slika 7.14b. Dekrementiramo remaining (primer 7.21b) in preidemo v naslednjo iteracijo.

Tretja iteracija ($i = 2$)**.** Ta iteracija je enaka prejšnji iteraciji. Inkrementiramo remaining in end (glej primer 7.22a), dodamo povezavo s pripono $\langle \mathsf{z} \rangle$ v $\mathcal{T}_P$ (slika 7.14b), dekrementiramo remaining (primer 7.22b) in preidemo v naslednjo iteracijo.

Četrta iteracija ($i = 3$)**.** Inkrementiramo remaining in end (primer 7.23a). Nato preverimo, ali iz trenutnega activeNode izhaja povezava z $J_i = \mathsf{x}$.

```
remaining = 1          remaining = 0
activeNode = root      activeNode = root
activeEdge = -1        activeEdge = -1
activeLenght = 0       activeLenght = 0
end = 1                end = 1
     (a)                    (b)
```

Primer 7.21: Stanje aktivne točke – druga iteracija



Slika 7.14: $\mathcal{T}_P$ po prvih dveh korakih

```
remaining = 1          remaining = 0
activeNode = root      activeNode = root
activeEdge = -1        activeEdge = -1
activeLenght = 0       activeLenght = 0
end = 2                end = 2
     (a)                    (b)
```

Primer 7.22: Stanje aktivne točke – tretja iteracija

Takšna povezava obstaja, zato postavimo activeEdge = 0 (0 zato, ker je to indeks v zaporedje J, iz katerega se začne pripona z x). Ujemanje je v dolžini enega znaka, zato postane activeLenght = 1. $\mathcal{T}_P$ postane nepopolno priponsko drevo. Ker nismo tvorili nobene nove povezave, ostane remaining = 1 (glej primer 7.23b).

```
remaining = 1          remaining = 1
activeNode = root      activeNode = root
activeEdge = -1        activeEdge = 0
activeLenght = 0       activeLenght = 1
end = 3                end = 3
     (a)                    (b)
```

Primer 7.23: Stanje aktivne točke – četrta iteracija

Peta iteracija ($i = 4$)**.** Kot vedno inkrementiramo remaining in end (glej primer 7.24a), nato začnemo obdelavo aktivne točke. Iz activeNote = root gremo v smeri activeEdge = 0, ki predstavlja pripono, ki se začne na indeksu $J_{activeEdge=0}$ = x. Po tej priponi se premaknemo za activeLenght = 1. Preverimo, ali je znak, ki smo ga dosegli, y. Ker je, povečamo activeLenght za 1 (primer 7.24b) in zaključimo iteracijo.

```
remaining = 2          remaining = 2
activeNode = root      activeNode = root
activeEdge = 0         activeEdge = 0
activeLenght = 1       activeLenght = 2
end = 4                end = 4
     (a)                    (b)
```

Primer 7.24: Stanje aktivne točke – peta iteracija

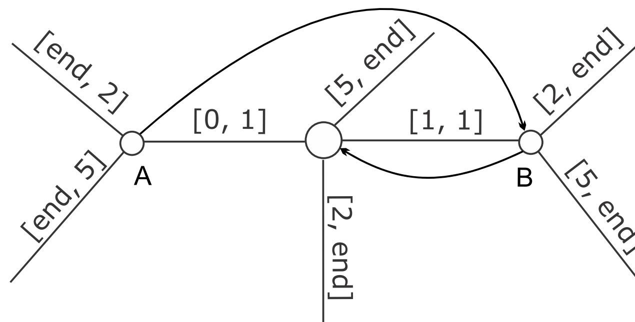
Šesta iteracija ($i = 5$)**.** Inkrementiramo remaining in end (primer 7.25a). remaining = 3, kar pomeni, da v $\mathcal{T}_P$ manjkajo že tri povezave. Nato se premaknemo od activeNode = root v activeEdge = 0, ki izbere povezavo s prvim znakom x, po kateri se premaknemo za activeLenght = 2 mesti, da dosežemo znak y prve pripone. Preverimo, ali je naslednji simbol a. Ker ni, tvorimo interno vozlišče A. Ustvarili smo notranje vozlišče, zato tvorimo priponski kazalec, ki kaže na koren. Ker smo ustvarili novo povezavo, zmanjšamo remaining, zmanjšamo activeLenght za 1 in povečamo activeEdge na 1 (primer 7.25b). S tem se premaknemo na naslednjo povezavo, ki izhaja iz korena s prvim znakom y. Tudi tokrat kreiramo notranje vozlišče (imenujmo ga B), iz katerega izhajata priponi ⟨yzxya⟩ in ⟨ya⟩. Priponski kazalec iz predhodno kreiranega notranjega vozlišča A, ki je kazal na root, premaknemo na vozlišče B, priponski kazalec iz B pa postavimo na root. Ponovno ažuriramo aktivno točko: remaining dekrementiramo, activeEdge povečamo na 2, activeLenght zmanjšamo na 0 (primer 7.25c). Ker je activeLenght = 0, kreiramo novo povezavo iz korena $\mathcal{T}_P$, ki hrani pripono ⟨a⟩. Dekrementiramo remaining in zaključimo iteracijo (stanje aktivne točke kaže primer 7.25d), $\mathcal{T}_P$ pa postane priponsko drevo, ki ga vidimo na sliki 7.15.

Sedma iteracija ($i = 6$)**.** Inkrementiramo remaining in end (primer 7.26a). Ker iz korena izhaja pripona s prvim znakom J_6 = x, postavimo activeEdge = 0 in activeLenght = 1 ter zaključimo. Stanje aktivne točke kaže primer 7.26b.

`remaining = 3`
`activeNode = root`
`activeEdge = 0`
`activeLenght = 2`
`end = 5`
(a)
	`remaining = 2`
	`activeNode = root`
	`activeEdge = 1`
	`activeLenght = 1`
	`end = 5`
	(b)

`remaining = 1`
`activeNode = root`
`activeEdge = 2`
`activeLenght = 0`
`end = 5`
(c)
	`remaining = 0`
	`activeNode = root`
	`activeEdge = 2`
	`activeLenght = 0`
	`end = 5`
	(d)

Primer 7.25: Stanje aktivne točke – šesta iteracija



Slika 7.15: $\mathcal{T}_P$ po prvih šestih korakih

`remaining = 1`
`activeNode = root`
`activeEdge = 2`
`activeLenght = 0`
`end = 6`
(a)
	`remaining = 1`
	`activeNode = root`
	`activeEdge = 0`
	`activeLenght = 1`
	`end = 6`
	(b)

Primer 7.26: Stanje aktivne točke – sedma iteracija

Osma iteracija ($i = 7$)**.** Inkrementiramo remaining in end (primer 7.27a). Preverimo, ali je v povezavi activeEdge = 0 (ta, ki ima prvi znak pripone x na položaju activeLenght + 1 = 2) znak y. Ker obstaja, povečamo activeLenght za 1 (primer 7.27b). $\mathcal{T}_P$ je še vedno takšno,

kot je na sliki 7.15.

```
remaining = 2           remaining = 2
activeNode = root       activeNode = root
activeEdge = 0          activeEdge = 0
activeLenght = 1        activeLenght = 2
end = 7                 end = 7
     (a)                     (b)
```

Primer 7.27: Stanje aktivne točke – osma iteracija

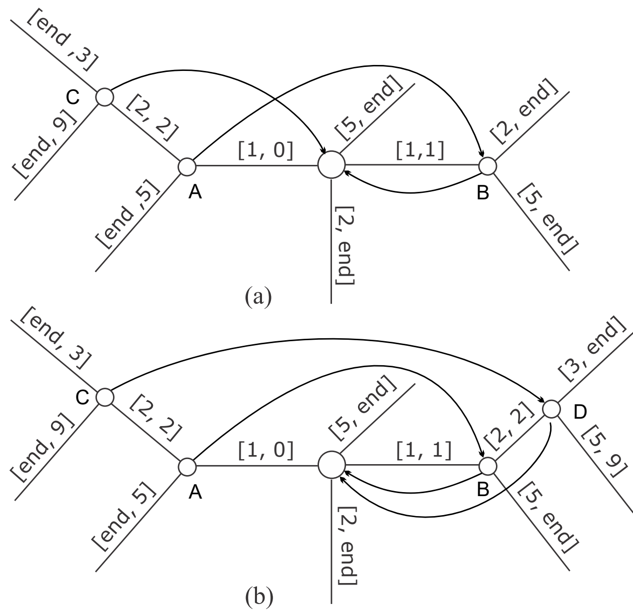
Deveta iteracija ($i = 8$)**.** Ko smo povečali remaining in end (primer 7.28a), preverimo, ali je v povezavi activeEdge naslednji znak od kazalca activeLength = z. Znak z dejansko obstaja, vendar za notranjim vozliščem A. To vozlišče postane aktivno vozlišče, aktivna povezava postane 2 (pripona ⟨zxyaxyz⟩), ujemanje pa je v dolžini enega znaka, zato je activeLenght = 1 (novo stanje aktivne točke vidimo kaže primer 7.28b). $\mathcal{T}_P$ se tudi tokrat ni spremenilo (vidimo ga na 7.15).

```
remaining = 3           remaining = 3
activeNode = root       activeNode = A
activeEdge = 0          activeEdge = 2
activeLenght = 2        activeLenght = 1
end = 8                 end = 8
     (a)                     (b)
```

Primer 7.28: Stanje aktivne točke – deveta iteracija

Deseta iteracija ($i = 9$)**.** Spet najprej inkrementiramo remaining in end. Ker je remaining = 4, vemo, da $\mathcal{T}_P$ manjkajo štiri povezave, da bi postalo priponsko drevo. Znak, ki ga obdelujemo, je stražar $\sigma_s =!$, torej znak, ki ga še nismo videli. Začnemo s podatki v aktivni točki (primer 7.29a). Najprej preverimo, ali je v priponi, kamor kažejo activeNode, activeEdge in activeLenght naslednji znak !. Ker ni, kreiramo notranje vozlišče C. Ustvarimo priponski kazalec, ki kaže na koren (slika 7.16a). Zmanjšamo remaning. Iz activeNode = A se po njegovem priponskem kazalcu postavimo v vozlišče B. Iz B se pomaknemo v smeri ActiveEdge = 2, to je v smeri pripone ⟨zxyaxyz⟩, in ker je activeLenght = 1, se postavimo na prvi znak, to je na znak z (primer 7.29b). Ker naslednji znak ni !, vstavimo novo notranje vozlišče

D, iz katerega izhajata dve povezavi. Postavimo še priponske kazalce. Na vozlišče D kaže priponski kazalec iz vozlišča C (prej smo ga postavili na koren), kazalec iz D pa kaže na koren. Trenutno situacijo kaže slika 7.16b. Zmanjšamo remaining, nato sledimo priponskemu kazalcu



Slika 7.16: $\mathcal{T}_P$ po desetem koraku – prvi del

vozlišča B, ki kaže na koren (primer 7.29c). Iz korena sledimo activeEdge, ki nas vodi do pripone ⟨zxyaxyz⟩ in se premaknemo na prvi znak (activeLenght = 1), ki je z. Naslednji znak ni !, zato vrinemo notranje vozlišče E. Priponski kazalec iz D postavimo na E, ki kaže na koren (slika 7.17a). Zmanjšamo remaining na 1. Ker nismo spremenili activeNode, dekrementiramo activeEdge in activeLenght (primer 7.29d). Ker je activeLenght = 0, iz korena tvorimo še povezavo za ! (situacijo kaže slika 7.17b). Nato dekrementiramo remaining, ki postane 0, zato zaključimo (primer 7.29e).

Finalizacija. V zadnjem koraku moramo $\mathcal{T}_P$ prirediti vrednosti v listih (listov na naših slikah do zdaj nismo označevali) z indeksi pripon v polje J. V vsaki povezavi $\mathcal{T}_P$ preštejemo število znakov od lista do drevesa ter to vrednost odštejemo od m. Za primer 7.18 je $m = |J| = 10$. Znakov ni treba šteti, seštejemo razlike indeksov na povezavah od lista do korena. Priponsko drevo, opremljeno z indeksi v polje J, kaže

```
remaining = 4          remaining = 3          remaining = 2
activeNode = A         activeNode = B         activeNode = root
activeEdge = 2         activeEdge = 2         activeEdge = 2
activeLenght = 1       activeLenght = 1       activeLenght = 1
end = 9                end = 9                end = 9

     (a)                    (b)                    (c)

remaining = 1          remaining = 0
activeNode = root      activeNode = root
activeEdge = 1         activeEdge = 1
activeLenght = 0       activeLenght = 0
end = 9                end = 9
     (d)                    (e)
```

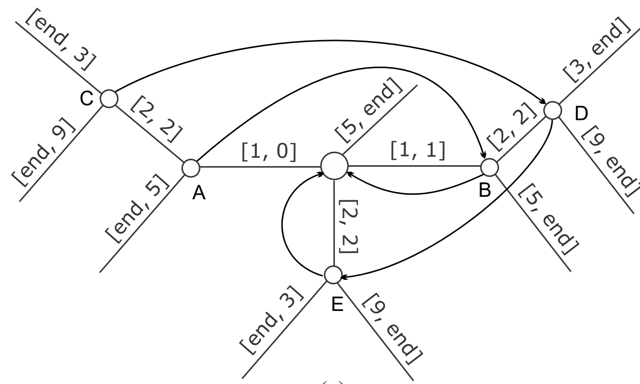
Primer 7.29: Stanje aktivne točke – deseta iteracija

slika 7.18

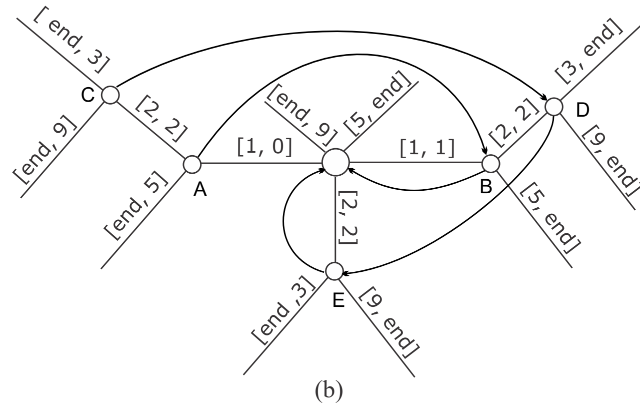
7.2.4 Uporaba priponskih dreves

Z zgrajenim priponskim drevesom lahko učinkovito rešimo različne naloge. V nadaljevanju si bomo nekatere pogledali.

1. **Ali je vzorec** p **del zaporedja** J? S to nalogo smo se spoprijeli v poglavju 2. Že takrat smo omenili, da bomo za primere, ko se zaporedje ne spreminja, spoznali druge pristope. Priponsko drevo je odlično orodje za ta namen. Poglejmo si primer za zaporedje $J = \langle \mathsf{rabarbara!} \rangle$, za katerega smo že sestavili priponsko drevo na sliki 7.5. Naj bo vzorec $p = \langle \mathsf{arb} \rangle$, za katerega nas zanima, ali je v zaporedju I oziroma J. V korenu drevesa preverimo, ali obstaja povezava z zaporedjem, ki se začne s črko a. Takšna povezava obstaja, zato se premaknemo po povezavi v vozlišče in iščemo povezavo, ki se začne s črko r. Tudi ta povezava obstaja, zato se po njej spustimo do naslednjega vozlišča in preverimo, ali v tem vozlišču obstaja povezava s prvo črko oznake b. Tudi takšna povezava obstaja, zato sporočimo, da iskani vzorec v zaporedju obstaja. Poglejmo še, ali v I obstaja vzorec $p = \langle \mathsf{baraba} \rangle$. Iz korena $\mathcal{T}_P$ najdemo povezavo s prvo črko b. Naslednji črki iz oznake na povezavi sta skladni z iskanim vzorcem. Dosežemo novo notranje vozlišče in preverimo, ali iz njega izhaja povezava s prvo črko a. Takšna

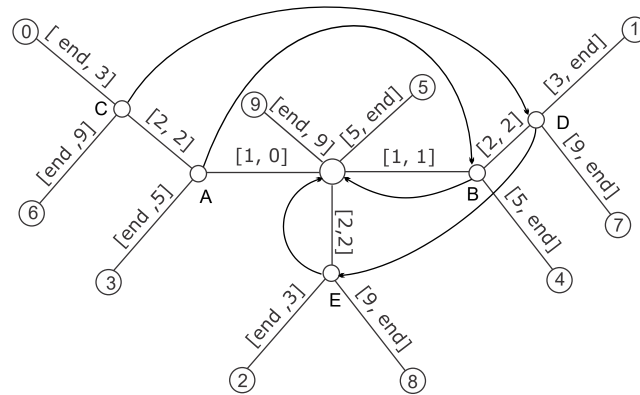


(a)



(b)

Slika 7.17: $\mathcal{T}_P$ po desetem koraku – drugi del



Slika 7.18: Končno priponsko drevo

povezava obstaja, zato preverimo v tej povezavi naslednji znak, če je b. Ker ! $\neq$ b, sporočimo, da vzorec v zaporedju ne obstaja.

2. **Ali je vzorec** p **pripona v** J? Iskanje odgovora je popolnoma enako kot v prejšnji točki, le da moramo najti ujemanje p do stražarja v $\mathcal{T}_P$.

3. **Kolikokrat se vzorec** P **ponovi v zaporedju** J? Naj bo $p =$⟨ra⟩ in zaporedje $J =$⟨rabarbara!⟩. V priponskem drevesu s slike 7.5 ugotovimo, da P obstaja (potujemo od korena po povezavah z oznako r in a). Iz vozlišča nato izhajata dve povezavi, zato velja, da sta v J dva vzorca P. Indeksa v listih, ki izhajata iz vozlišča, povesta, kje P v J dejansko je. V našem primeru najdemo indeksa 0 in 7. To sta tudi položaja začetka P v zaporedju J.

4. **Najdaljše skupno podzaporedje.** Naj bosta zaporedji I_1 in I_2. Poiščimo najdaljše skupno podzaporedje (angl. the longest common substring, LCS). Najprej tvorimo novo skupno zaporedje $J = I_1\#I_2!$, kjer sta # in ! stražarja. Nato zgradimo priponsko drevo nad zaporedjem J. Takšnemu priponskemu drevesu pravimo tudi **posplošeno priponsko drevo** (angl. generalized suffix tree).

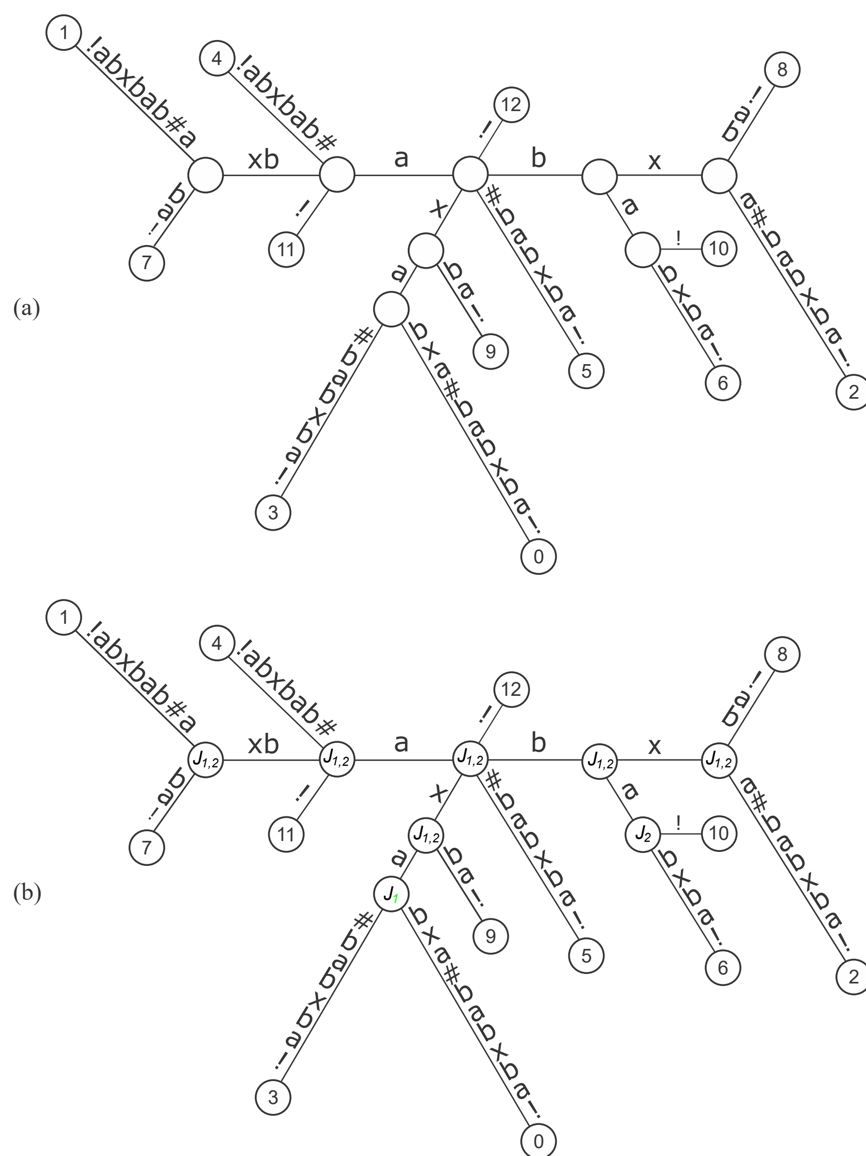
$J_1 = \langle$`xabxa!`$\rangle$
$J_2 = \langle$`babxba!`$\rangle$

$J = \langle$`xabxa#babxba!`$\rangle$

Primer 7.30: Iskanje najdaljšega skupnega podzaporedja

Priponsko drevo za podzaporedje J iz primera 7.30 vidimo na sliki 7.19a. Listi $[0, 4]$ so pripone zaporedja J_1, listi $[6, 11]$ pa pripone zaporedja J_2. Najprej opravimo obhod notranjih vozlišč drevesa in označimo, ali hranijo pripone zaporedja J_1, pripone zaporedja J_2 ali pripone obeh, to je $J_{1,2}$ na sliki 7.19b. Zdaj poiščemo najgloblje vozlišče, ki je označeno z $J_{1,2}$. V našem primeru imamo dve takšni vozlišči, eno po povezavi a in drugo po povezavi b. Primerjamo dolžine zlepljenih oznak v obeh povezavah. V povezavi a je dolžina 3 (zaporedje ⟨abx⟩), v povezavi b pa 2; zaporedje ⟨bx⟩). Najdaljše skupno podzaporedje je zato $LCS(J_1, J_2) =$ ⟨abx⟩.

Razmislimo še o časovni zahtevnosti tega algoritma. Naj bo $m_1 = |J_1|, m_2 = |J_2|$ in $m = m_1 + m_2$. Z Ukkonenovim algoritmom zgradimo priponsko drevo v linearnem času $O(m)$. Obhod v drevesu prav tako



Slika 7.19: Najdaljše skupno podzaporedje

opravimo v linearnem času. Skupna časovna zahtevnost algoritma je torej $O(m)$.

Naloge

1. Z Mamber-Mayersovim postopkom sestavite priponsko polje za zaporedje $I = \langle\textsf{vrane družijo se rade}\rangle$. Priponsko polje uporabite za konstrukcijo Burrows-Wheelerjeve transformacije. Pravilnost rezultata preverite z izvornim postopkom BWT.
2. S primerom razložite pravili P1 in P2 algoritma DC3.
3. Sestavite številsko drevo za zaporedje $I = \langle\textsf{regaregakvak}\rangle$. Ne pozabite dodati stražarja.
4. Napišite algoritem, ki bo sestavil številsko drevo. Razmislite o njegovi vizualizaciji, pri čemer se omejite na največ 30 znakov v zaporedju I.
5. Za zaporedje $I = \langle\textsf{rabarbara}\rangle$ narišite priponsko drevo. Ne pozabite dodati stražarja in priponskih kazalcev. Pravilnost rezultata lahko preverite v [77].
6. Napišite algoritem, ki bo z naivno metodo sestavil priponsko drevo. Preverite, ali je hitrost konstrukcije drevesa odvisna od zaporedja in velikosti abecede.
7. Razložite pravila, ki jih uporablja Ukkonenov algoritem.
8. Kakšna je razlika med nepopolnim priponskim drevesom in priponskim drevesom?
9. S primerom zaporedja $I = \langle\textsf{Jagababa}\rangle$ razložite Ukkonenov postopek gradnje priponskega drevesa.
10. Razložite pomen spremenljivk v aktivni točki pri Ukkonenovem algoritmu.
11. Demonstrirajte delovanje Ukkonenovega algoritma za zaporedje $I = \langle\textsf{gorinagorigori}\rangle$.
12. Za zaporedji $I_1 = \langle\textsf{Prestreljenik}\rangle$ in $I_2 = \langle\textsf{strelec}\rangle$ poiščite najdaljše skupno podzaporedje s pomočjo priponskega drevesa.

Poglavje 8

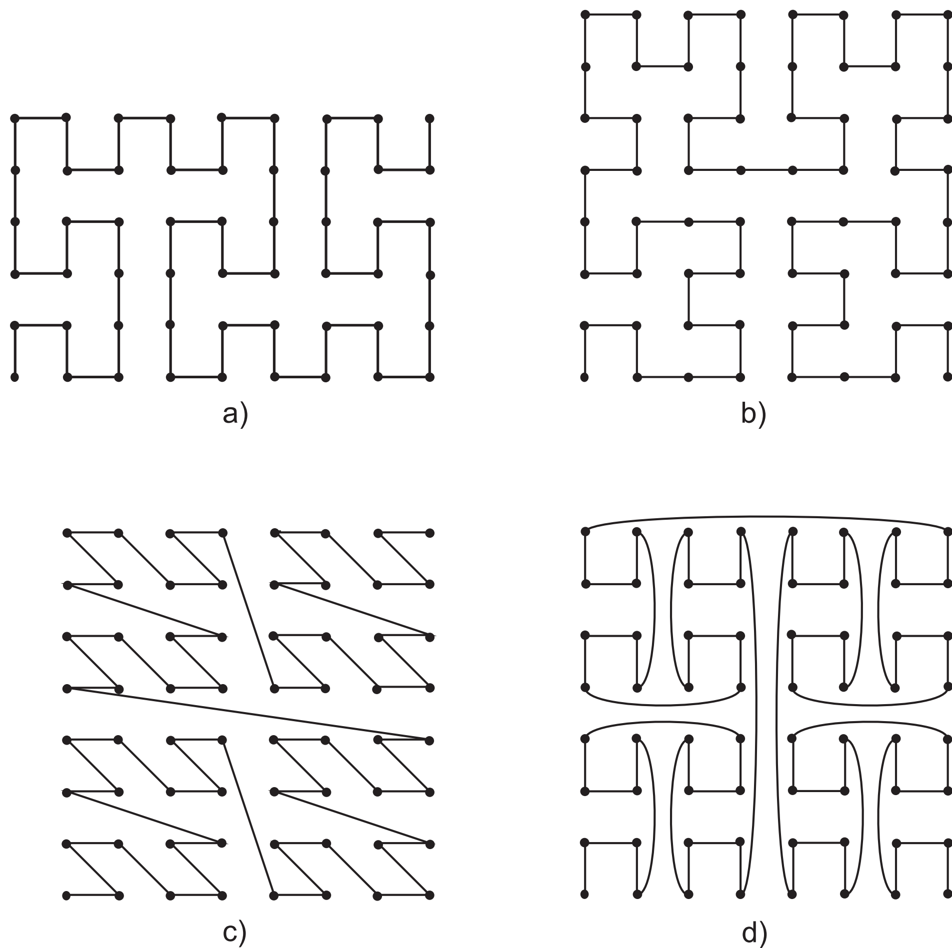
Pretvorba večdimenzionalnih podatkov v zaporedja

Na prvi pogled je videti, da s tem poglavjem izstopamo iz konteksta tega gradiva, saj smo do zdaj vedno obravnavali samo enodimenzionalne podatke, to so zaporedja. Namen tega poglavja je pokazati, da je možno preslikati podatke iz večdimenzionalnega prostora (omejili se bomo samo na 2D rastrski prostor) v zaporedje. Dejansko je to celo zelo pogost pristop, saj je zaporedje najbolj naravna organizacija podatkov v pomnilniku, na pomnilnih medijih in v podatkovnih zbirkah [78]. Ko znamo pretvoriti večdimenzionalne podatke v zaporedje, lahko nad njimi uporabimo tudi algoritme, ki smo jih do zdaj spoznali. V tem poglavju si bomo ogledali dve možnosti, in sicer krivulje polnjenja prostora in opis rasteriziranih geometrijskih objektov z verižnimi kodami.

8.1 Krivulje polnjenja prostora

Če želimo učinkovito iskati večdimenzionalne podatke, ki so shranjeni v linearnih podatkovnih strukturah, na primer v podatkovni zbirki, moramo uvesti preslikavo iz večdimenzionalnega prostora v enodimenzionalnega. Pravimo, da podatkom priredimo **prostorski indeks** (angl. spatial index) ali krajše kar indeks. Večdimenzionalne podatke si lahko predstavljamo kot točke v prostoru, skozi katere želimo potegniti krivuljo tako, da ta vsako točko obišče natanko enkrat. Takšnim krivuljam pravimo **krivulje polnjenja prostora** (angl. space filling curves, SFC). Prvo krivuljo SFC je izumil Giuseppe Peano leta 1890. David Hilbert je le eno leto pozneje podal geometrični opis drugačne krivulje polnjenja prostora. Veliko pozneje, leta

1966 [79], je Morton predstavil Z-urejeno krivuljo (angl. Z-order curve) ali Mortonovo krivuljo, Grayeva krivulja pa temelji na Grayevi kodi [80] (glej sliko 8.1). Obstaja še množica drugih krivulj polnjenja prostora (krivulja Gosperja, Kocha, Moorea, Sierpińskega). Matematične definicije krivulj polnjenja prostora najdemo v številnih virih [78, 81, 83], nas pa bo zanimala predvsem njihova uporaba. V nadaljevanju se bomo osredotočili na Hilbertovo krivuljo polnjenja prostora, ki je pogosto našla uporabo v računalniških aplikacijah [84, 85, 86, 87].



Slika 8.1: Krivulje polnjenja prostora v 2D: (a) Peanova, (b) Hilbertova, (c) Mortonova in (d) Grayeva

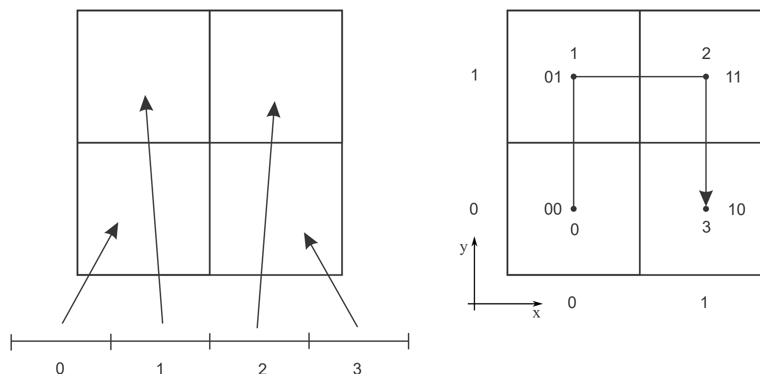
8.2 Konstrukcija Hilbertove krivulje

Preden si pogledamo proces konstrukcije Hilbertove krivulje, pojasnimo pojma red in aproksimacija krivulje.

- **Red krivulje** je število korakov oziroma iteracij, ki smo ga opravili v postopku tvorbe SFC.
- Hilbertova krivulja nekega končnega reda je **aproksimacija** Hilbertove krivulje polnjenja prostora, saj ne poteka skozi vsako točko prostora, ampak samo skozi središčne točke končnega števila enako velikih celic, katerih unija pokrije celotni prostor. V ravnini je celica kvadratna, prostor pa je enotski kvadrat.

Konstrukcijo Hilbertove krivulje opravimo v naslednjih korakih:

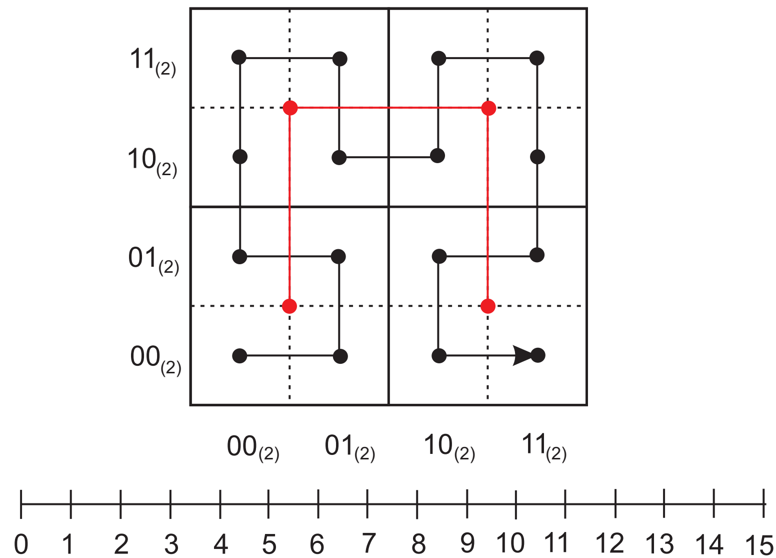
- **Konstrukcija krivulje reda 1.** Enotski interval in enotsko celico razdelimo na četrtine. Vsak podinterval nato priredimo celici na način, da si celice, ki so preslikane iz sosednjih podintervalov, delijo rob (slika 8.2a), s čimer vzpostavimo urejenost med celicami. Krivulja, ki jo potegnemo skozi središče celic, je Hilbertova krivulja prvega reda (slika 8.2b). Uvedimo binarno označevanje celic na osi x in y ter določimo zaporedne binarne kode zaporednim točkam krivulje. Opazimo, da se kode koordinat celic in vrstnega reda točk ne ujemajo popolnoma. To bomo obravnavali nekoliko pozneje.



Slika 8.2: Konstrukcija Hilbertove krivulje 1. reda

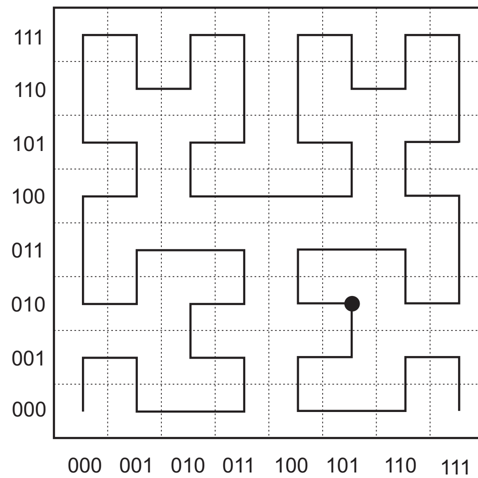
- **Konstrukcija krivulje reda 2.** Proces delitve ponovimo za vsak podinterval in vsako celico. Rezultat so štiri skupine enako velikih podintervalov in celic. Med njimi ponovno vzpostavimo preslikavo med

podintervali in celicami, kot smo to storili pri prvem koraku. Tudi tokrat pazimo, da si celice delijo robove tako, da ohranjamo sosednost preslikav iz intervalov. Na sliki 8.3 vidimo rezultat – Hilbertovo krivuljo drugega reda. Opazovanje nas vodi do splošnega pravila za konstrukcijo Hilbertove krivulje poljubnega reda.



Slika 8.3: Konstrukcija Hilbertove krivulje 2. reda narisana s črno; krivulja prvega reda je rdeča

- **Konstrukcija krivulje reda** k. Krivuljo reda k konstruiramo tako, da razdelimo vse celice v štiri manjše celice, nato pa zamenjamo vsako točko na krivulji reda $k-1$ s krivuljo prvega reda. Ob opazovanju slik 8.2 in 8.3 ugotovimo, da krivulja reda k sestoji iz štirih krivulj reda $k-1$, pri čemer imata prva in četrta krivulja različno orientacijo s, druga in tretja pa enako orientacijo kot krivulja reda $k-1$. Vstavljene krivulje reda $k-1$ po rotaciji povežemo tako, da je zadnja točka trenutne krivulje povezana s prvo točko naslednje krivulje. Razdalja med vsakim takšnim parom točk je enaka razdalji med katero koli drugo točko na krivulji reda k. Hilbertovo krivuljo reda $k = 3$ kaže slika 8.4.



Slika 8.4: Konstrukcija Hilbertove krivulje 3. reda

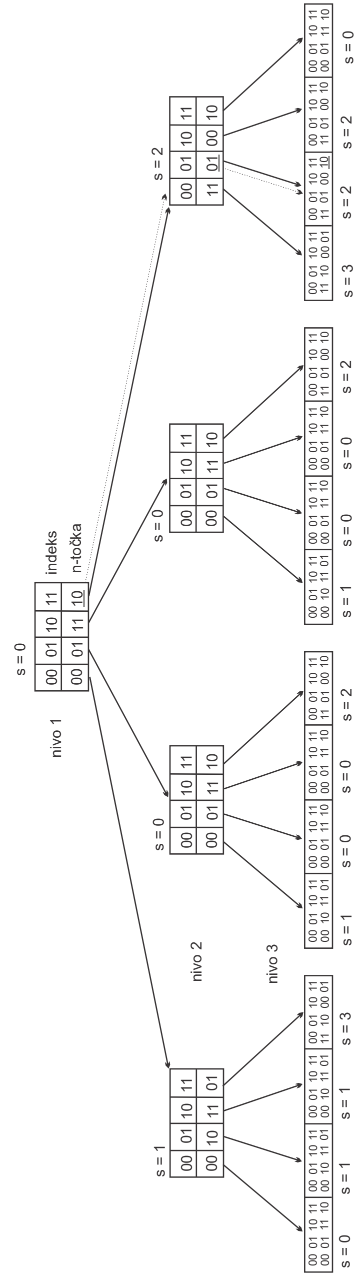
8.2.1 Predstavitev Hilbertove krivulje z drevesom

Proces rekurzivne delitve lahko predstavimo tudi z drevesom, ki ponazarja proces preslikave in omogoča zasnovo algoritmov za prostorsko iskanje. Drevo je uravnoteženo in njegova višina je enaka redu krivulje. Pri razlagi si bomo pomagali z dvema pojmoma:

- **n-točka** (angl. n-point) je zaporedje zlepljenih bitnih kod, sestavljenih iz enakoležnih bitov koordinate točke, ki leži na Hilbertovi krivulji in ustreza središču celice.
- **Indeks** (tudi ključ ali dolžina, angl. derived key) predstavlja število točk, skozi katere je potekala Hilbertova krivulja. Ključ je predstavljen z $2 \cdot k$-bitno vrednostjo.

Na sliki 8.2 so n-točke zaporedje koordinat točk Hilbertove krivulje $\langle 00, 01, 11, 10\rangle$, indeksi pa ustrezajo zaporedju intervalov $\langle 0, 1, 2, 3\rangle$ na številski premici oziroma njihovim binarnim vrednostim $\langle 00, 01, 10, 11\rangle$. V korenu drevesa predstavimo Hilbertovo krivuljo prvega reda tako, da zapišemo n-točke in indekse ter jih združimo v pare. Dobimo {[00], (00)}, {[01], (01)}, {[10], (11)} in {[11], (10)}, pri čemer je prva vrednost para indeks, označen z oglatim, druga vrednost pa n-točka, označena z okroglim oklepajem.

Rekurzivna konstrukcija krivulje drugega reda povzroči, da postane vsak par starš vozlišča, ki tudi vsebuje množico parov {(indeks, n-točka)}. Dve



Slika 8.5: Drevo Hilbertove krivulje 3. reda. Zaradi boljše berljivosti smo izpustili oklepaje pri n-točkah in indeksih

vozlišči potomcev hranita enako preslikavo kot njun starš, drugi dve pa različno, saj, kot smo že ugotovili, moramo opraviti rotacijo krivulje reda 1. Primer drevesa Hilbertove krivulje reda $k = 3$ vidimo na sliki 8.4.

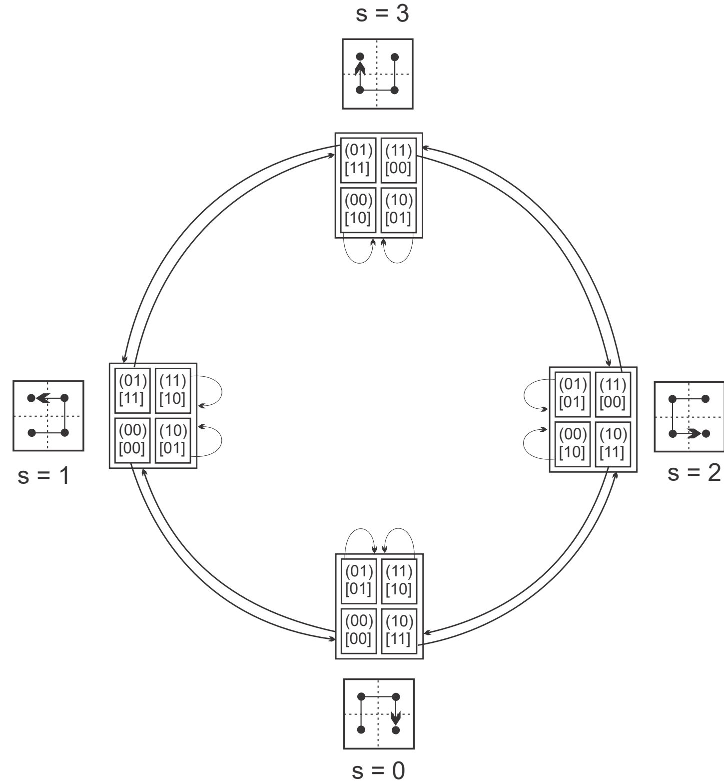
Ko smo konstruirali drevo, lahko iz koordinate točke določimo indeks D, ki ga na začetku inicializiramo kot $D = \{\}$. Vzemimo točko (5, 2), ki naj leži na Hilbertovi krivulji $k = 3$ (na sliki 8.4 je označena s krožcem). Koordinate najprej pretvorimo v binarne $(101_{(2)}, 010_{(2)})$, iz katerih tvorimo zaporedje n-točk $\langle(10),(01),(10)\rangle$. Preiskovanje drevesa s slike 8.5 in določitev indeksa začnemo v korenu drevesa, kjer n-točki (10) (podčrtan na sliki) ustreza indeks [11], ki ga shranimo v spremenljivko $d = 11$, $D = D \oplus^{1} d = 11$. Nato se po črtkani povezavi premaknemo na drugi nivo drevesa v skrajno desno vozlišče, na sliki 8.5 označenega kot $s = 2$. Vzamemo drugo n-točko (01) in preko nje določimo indeks $d = 01$. Z lepljenjem dobimo $D = 1101$. Nato napredujemo do zadnjega nivoja drevesa (sledimo črtkani puščici). n-točka je tokrat (10), kar nam da $d = 11$. Opravimo lepljenje $D = D \oplus d = 110111$. D je iskan indeks, ki predstavlja dolžino Hilbertove krivulje tretjega reda do točke (5, 2). Pravilnost ključa $D = 110111_{(2)} = 55_{(10)}$ lahko hitro preverimo na sliki 8.4 tako, da preštejemo število točk na Hilbertovi krivulji do te točke.

Po analognem postopku iz indeksa dobimo tudi koordinato iskane točke. A konstrukcija drevesa Hilbertovih stanj ni najelegantnejša rešitev. Druga, pogostejša realizacija preslikave med koordinatami točk in indeksom, je uporaba diagrama stanj.

8.3 Transformacijo v in iz prostora Hilbertove krivulje z diagramom stanj

Preslikavo med položajem na Hilbertovi krivulji in točko v prostoru ter obratno lahko realiziramo z drevesom, kot smo pravkar spoznali. Število tipov vozlišč v tem drevesu, to je orientacij Hilbertovih krivulj 1. reda, pa je končno, ne glede na globino drevesa. To lahko hitro preverimo iz grafične predstavitve krivulje na sliki 8.5. Zato lahko drevo predstavimo z diagramom stanj, kjer vsako vozlišče drevesa predstavlja stanje. Ko nivo drevesa preseže relativno nizek prag, je vozlišč več kot stanj, zato je diagram stanj precej kompaktnejša oblika. Z nekoliko premisleka bi lahko diagram stanj, kot ga vidimo na sliki 8.6, sestavili tudi sami. Diagram stanj je znan tudi za 3D, za višje dimenzije pa je algoritmičen postopek razvil Bially [88].

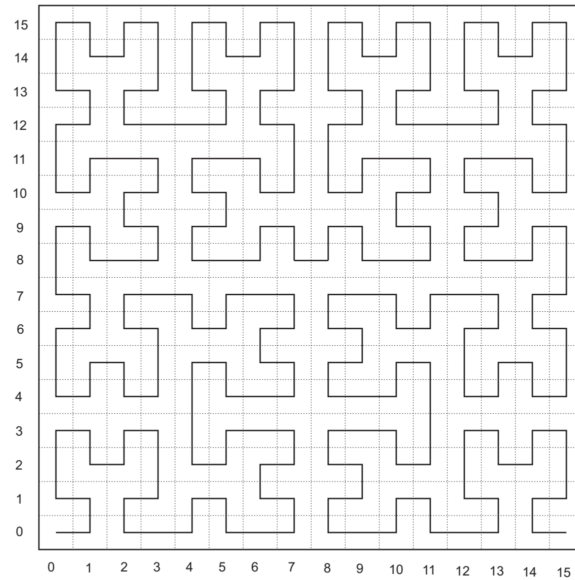
[1] Označuje operacijo lepljenja nizov.



Slika 8.6: Diagram stanj za 2D Hilbertovo krivuljo; n-točke so označene z (), indeksi pa z []

Poglejmo si primer uporabe diagrama stanj za Hilbertovo krivuljo tretjega reda in točko (5, 2) s slike 8.4. Tako kot v primeru drevesa tudi tokrat najprej določimo zaporedje n-točk $\langle(10),(01),(10)\rangle$. V diagram stanj vedno vstopimo v stanje 0. Poiščemo n-točko (10) in postavimo $D = d = 11$. Vidimo, da iz para {(10), [11]} izhaja puščica v stanje 2, zato se v to stanje tudi premaknemo in poiščemo n-točko (01). Ta nam da $d = 01$ in $D = 1101$. Kvadrant, kjer je (01), ima puščico, ki ne prehaja v drugo stanje, zato ostanemo v stanju 2. Zadnja n-točka je (10), ki nam v stanju 2 da $d = 11$. Dobili smo indeks $D = 110111_{(2)}$ – enako, kot v primeru uporabe drevesa.

Za vajo si poglejmo še primer Hilbertove krivulje reda 4 na sliki 8.7. Izberimo točko (9, 12) $= (1001_{(2)}, 1100_{(2)})$ ter tvorimo zaporedje n-točk $\langle(11),(01),(00),(10)\rangle$. Začnemo v stanju 0, poiščemo (11) in dobimo $D = d = [10]$. Ostanemo v stanju 0. Poiščemo (01). Dobimo $d = 01$ in $D \oplus d = 1001$. Ostanemo v stanju 0. Poiščemo (00). Dobimo $d = 00, D \oplus d =$



Slika 8.7: Hilbertova krivulja za $k = 4$

100100. Premaknemo se v stanje 1. (10) nam da $d = 01$, tako da dobimo $D = 10010001_{(2)} = 145_{(10)}$.

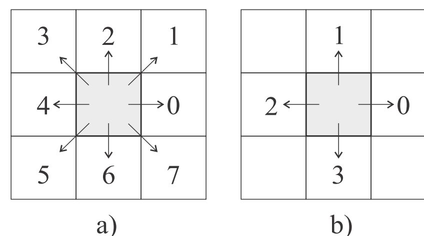
Poglejmo še zadnji primer za točko (3, 6) $= (0011_{(2)}, 0110_{(2)})$, ki tvori zaporedje n-točk $\langle$(00), (01), (11), (10)$\rangle$. Prvo n-točko poiščemo v stanju 0 in dobimo $D = 00$. Premaknemo se v stanje 1, kjer nam druga n-točka da $d = 11$ in $D = 0011$. Iz diagrama stanj ugotovimo, da se moramo premakniti v stanje 3, ker nam (11) da $d = 00$, tako da dobimo $D = 001100$. Tudi tokrat se moramo premakniti v naslednje stanje, to je stanje 2. Zadnja n-točka je (10), kar določi $D = 00110011$. Poglejmo še indeks D; $D = 00110011_{(2)} = 51_{(10)}$, kar lahko preverimo s štetjem obiskanih točk na Hilbertovi krivulji.

8.4 Verižne kode

Verižna koda (angl. chain code) je zaporedje ukazov, s katerimi se premikamo po robu geometrijskega objekta/regije. Obstajajo različne verižne kode [89], ki si jih bomo, hkrati z njihovimi lastnostmi, ogledali v tem podpoglavju.

8.4.1 Freemanova verižna koda v osem smeri

Freemanova verižna koda v osem smeri (angl. Freeman chain code in eight directions – F8) je najstarejša verižna koda. Predlagal jo je Freeman leta 1961 [90]. Iz referenčnega robnega piksla se lahko premaknemo v enega od osmih sosednjih pikslov, kot kaže slika 8.8a. Abeceda kode ima 8 simbolov σ_i: $\Sigma_{F8} = \{0, 1, 2, 3, 4, 5, 6, 7\}$.



Slika 8.8: Simboli verižne kode (a) F8 in (b) F4

Postopek generiranja verižne kode F8 je sestavljen iz naslednjih korakov: izberemo začetni robni piksel in shranimo njegove koordinate (x, y). Odločimo se za smer potovanja po robnih pikslih (sourna ali protiurna), nato pa skonstruiramo kodo tako, da se premikamo med sosednjimi robnimi piksli in zapišemo Freemanovo kodo premika. Kodo, ki se premika skozi središča pikslov, bomo imenovali **središčna verižna koda**. Lastnosti verižne kode F8 so naslednje:

- položaj objekta je odvisen le od koordinat začetne točke verižne kode (ta lastnost je dejansko skupna vsem verižnim kodam);

- povečevanje objekta za faktor s izvedemo tako, da vsak simbol F8 shranimo s-krat;

- zaporedje enakih vrednosti predstavlja zaporedje pikslov, ki si sledijo v ravni črti (vodoravni, navpični ali pod kotom 45°);

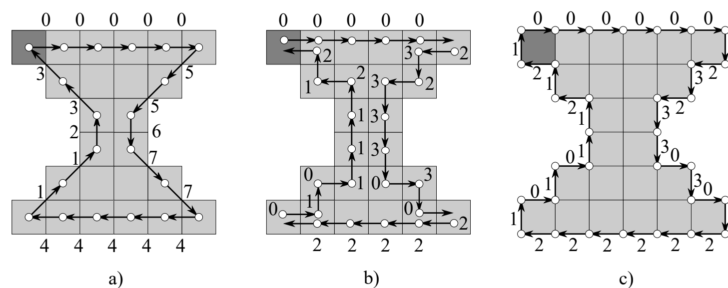
- če vsakemu simbolu $\sigma_i \in \Sigma_{F8}$ v zaporedju prištejemo vrednost naravnega števila n, $((\sigma_i + n) \mod 8)$, objekt zavrtimo za $n \times 45^o$ v smeri urinega kazalca;

- podobno, če vsaki vrednosti verižne kode odštejemo n, $((\sigma_i - n) \mod 8)$, objekt zavrtimo za $n \times 45^o$ v obratni smeri urinega kazalca.

8.4.2 Freemanova verižna koda v štiri smeri

Freemanova verižna koda v štiri smeri (angl. Freeman chain code in four directions – F4), ki jo je tudi predlagal Freeman, dovoli premik iz trenutnega piksla v sosednje piksle samo preko skupnih robov (slika 8.8b). Abeceda kode F4 je zato manjša: $\Sigma_{F4} = \{0, 1, 2, 3\}$. Konstrukcija kode F4 je enaka kot pri F8, tudi lastnosti so enake, le da lahko objekt zavrtimo za kote $\pm 90^o$ in 180°.

Kodo F4 lahko uporabimo na dva načina. Pri prvem načinu potujemo skozi središče pikslov, tako kot pri F8, pri drugem pa po mejnih robovih pikslov. Takšni kodi pravimo **lomna verižna koda** (angl. crack chain code, $F4^C$). Primer objekta, opisanega z različnimi Freemanovimi verižnimi kodami, vidimo na sliki 8.9.



Slika 8.9: Geometrijski objekt, opisan s (a) F8, (b) F4 in (c) $F4^C$

8.4.3 Izpeljanke Freemanove verižne kode

1. Freeman je pozneje predlagal **diferenčno vozliščno kodo** (angl. Chain-Difference Coding – CDC) [91]. Vsak mejni piksel p_i (razen prvega p_0 in drugega p_1) zakodiramo z relativno razliko kotov $\measuredangle(p_i - p_{i-1}, p_{i-1} - p_{i-2})$. Izkaže se, da so statistično najbolj verjetne razlike kotov 0°, ki jim sledijo koti $\pm 45^o$ (glej razpredelnico 8.1).

Razpredelnica 8.1: Verjetnost spremembe smeri kotov med sosednjimi piksli.

razlika kotov	0°	$\pm 45^o$	$\pm 90^o$	$\pm 135^o$	$\pm 180^o$
verjetnost	0.453	0.488	0.044	0.012	0.003

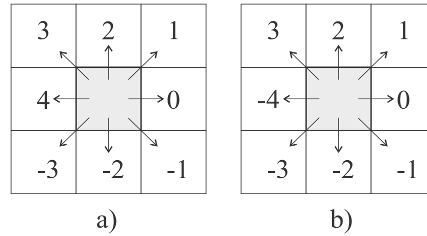
To lastnost sta izkoristila Liu in Žalik [92], ki sta predlagala eno prvih metod za stiskanje verižnih kod, temelječo na Huffmanovih kodah (glej razpredelnico 8.2).

Razpredelnica 8.2: Huffmanove kode za kodiranje razlike kotov

sprememba smeri	verjetnost	Huffmanova koda
0°	0,453	0
45°	0,244	10
-45°	0,244	110
90°	0,022	1110
-90°	0,022	11110
135°	0,006	111110
-135°	0,006	1111110
180°	0,003	1111111

2. Leta 1992 je Bribiesca predstavil izpeljanko verižne kode F8, ki jo imenujemo **usmerjena 8-smerna Freemanova verižna koda** (angl. Directional Freeman Chain Code of eight directions, DF8) [93]. Bribiesca je spremenil abecedo tako, da je simbole 5, 6 in 7 zamenjal z vrednostmi -3, -2 in -1. Če je orientacija verižne kode sourna, tudi simbol 4 nadomestimo z vrednostjo -4 (slika 8.10). S to spremembo preprosto ugotovimo, ali verižna koda predstavlja sklenjen objekt. Če je vsota vrednosti uporabljenih verižnih kod 8 pri protiurni orientaciji oz. -8 pri sourni, je verižna koda sklenjena, sicer ni.

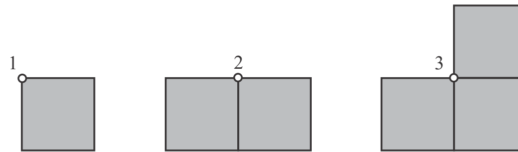
3. Različico verižne kode F4 je predstavil Nunes s sodelavci [94]. Predlagali so kodiranje relativne spremembe obhoda, kodo pa poimenovali **verižna koda razlik** (angl. Differential Chain Code, DCC). Abeceda DCC ima samo 3 simbole, $\Sigma_{DCC} = \{R, L, S\}$, kjer R pomeni zasuk v desno, L, zasuk v levo in S nadaljuj v isti smeri. Koda DCC je lomna koda.



Slika 8.10: Kode DF8 pri protiurni (a) in sourni (b) orientaciji

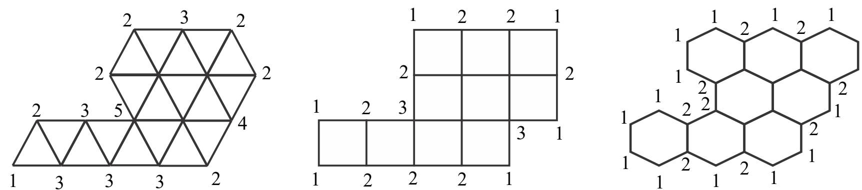
8.4.4 Ogliščna verižna koda

Leta 1999 je Bribiesca izumil zanimivo verižno kodo, ki jo imenujemo ogliščna verižna koda (angl. vertex chain code – VCC) [95]. Koda VCC je definirana s številom robnih pikslov objekta v opazovanem oglišču, kot vidimo na sliki 8.11. Abeceda vozliščne kode VCC sestoji samo iz treh simbolov, $\Sigma_{VCC} = \{1, 2, 3\}$. VCC je lomna vozliščna koda in se vedno sklene. Zaporedje simbolov $\langle 1, 1, 1, 1 \rangle$ opiše en piksel.



Slika 8.11: Simboli vozliščne kode VCC

Koda VCC je uporabna tudi, ko celice niso štirikotniki [95]. Primer vidimo na sliki 8.12. Ugotovimo, da se v tem primeru spremeni tudi abeceda. Seveda bomo v nadaljevanju privzeli, da so celice piksli pravokotne/kvadratne oblike.



Slika 8.12: Koda VCC pri celicah različnih oblik

Verižna koda VCC ima naslednje lastnosti:

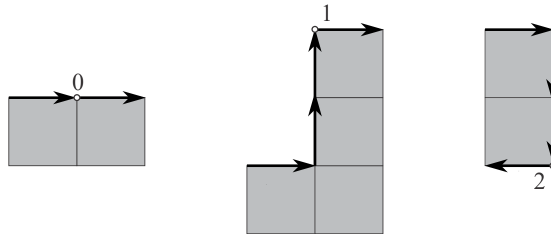
- ravno (navpično ali vodoravno) črto opisuje zaporedje simbolov 2;

- diagonalno črto pod kotom $\pm 45^o$ označuje izmenjujoče se zaporedje simbolov $\langle 13 \rangle$ ali $\langle 31 \rangle$;
- koda VCC je neodvisna glede na vrtenje (za $\pm 90^o$ in 180°) in zrcaljenje;
- kodo VCC lahko normaliziramo; simbole vrtimo tako dolgo, da zaporedje simbolov predstavlja število z najmanjšo vrednostjo, kot vidimo pri naslednjem primeru:
  $\langle 1311232121321213113312 \rangle$ – zasuk v desno
  $\langle 3112321213212131133121 \rangle$ – zasuk v desno
  $\langle 1123212132121311331213 \rangle$ – normalizirana koda
- kodo VCC lahko enostavno stisnemo; ker je, statistično, najpogostejši simbol VCC 2, ga predstavimo z enim bitom, preostala simbola pa z dvema.

8.4.5 Triortogonalna verižna koda

Triortogonalno verižno kodo (angl. Three OrThogonal chain code – 3OT) sta predstavila Sáchez-Cruz and Rodríguez-Diaz [96]. Tudi ta koda ima abecedo, predstavljeno iz treh simbolov, $\Sigma_{3OT} = \{0, 1, 2\}$, katerih pomen je naslednji (glej sliko 8.13):

- če je smer premika enaka, kot je bila smer premika predhodnega piksla, je koda 0;
- če je trenutna smer enaka smeri predhodnika, katerega koda je različna od 0, potem je koda 1;
- sicer je koda 2.



Slika 8.13: Simboli verižne kode 3OT

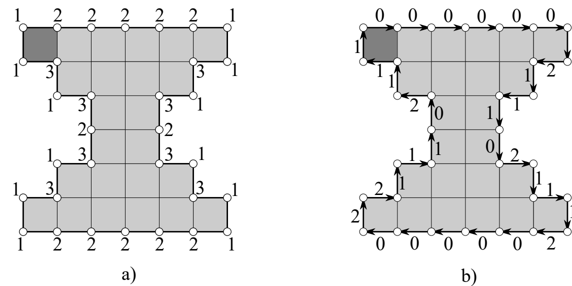
3OT je lomna koda, katere postopek konstrukcije je naslednji:

- izberemo ekstremno oglišče (na primer, zgoraj levo);
- izberemo smer obhoda;
- prvi rob, ki ni vodoraven, dobi kodo 1;
- preostale robove zakodiramo po zgornjem postopku.

Lastnosti kode 3OT so naslednje:

- ravno (navpično ali vodoravno) črto označuje zaporedje simbolov 0;
- diagonalno črto pod kotom $\pm 45^o$ predstavlja zaporedje simbolov 1;
- koda 3OT je neodvisna glede na vrtenje (za $\pm 90^o$ in 180°) in zrcaljenje.

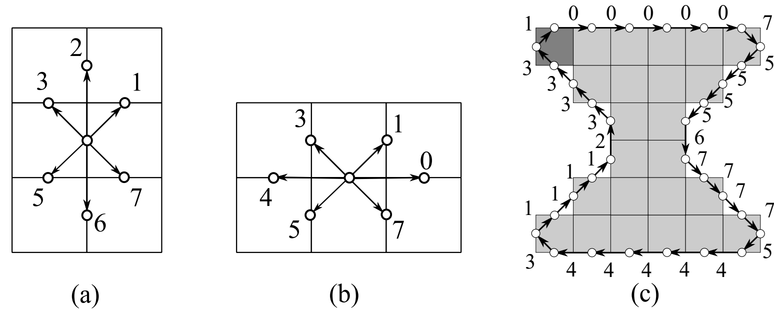
Slika 8.14 kaže geometrijski objekt z označenimi kodami VCC in 3OT.



Slika 8.14: Geometrijski objekt, opisan z (a) VCC in (b) 3OT

8.4.6 Središčno-lomna verižna koda

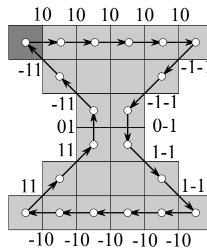
Središčno-lomno verižno kodo (angl. Mid-Crack Chain Code – MCCC) sta predlagala Shih in Wong [97] s ciljem bolje oceniti obseg in ploščino rasteriziranega objekta. MCCC kombinira središčno verižno kodo F8 z lomno kodo F4, pri tem pa povezuje središčne točke robov pikslov. Abeceda $\Sigma_{MCCC} = \Sigma_{F8}$. Pri navpičnih robovih pikslov ne dovolimo simbolov 0 in 4 (slika 8.15a), pri vodoravnih pa ne 2 in 6 (slika 8.15b). Opis objekta z verižno kodo MC vidimo na sliki 8.16c. Algoritem za konstrukcijo verižne kode je podrobno opisan v [98]. Lastnosti središčno-lomne verižne kode so enake kodi F8.



Slika 8.15: (a, b) Simboli središčno-lomne verižne kode, (c) opis objekta z zaporedjem središčno-lomnih kod

8.4.7 Nepredznačena verižna koda Manhattan

Nepredznačena verižna koda Manhattan (angl. Unsigned Manhattan Chain Code – UMCC) je predstavljena v [99]. Koda opisuje premikanje po robnih pikslih ločeno v smeri x in y. Če od dveh zaporednih robnih pikslov odštejemo njuni koordinati x in y, dobimo predznačeno verižno kodo Manhattan (MCC), katere abeceda je $\Sigma_{MCC} = \{-1, 0, 1\}$. Na sliki 8.16 vidimo primer opisa meje geometrijskega objekta z MCC, kode pa povzema razpredelnica 8.3:



Slika 8.16: Geometrijski objekt, opisan s predznačeno verižno kodo Manhattan

Razpredelnica 8.3: Predznačena verižna koda Manhattan za objekt s slike 8.16

$$\begin{array}{c|cccccccccccccccccccc} \mathrm{MCC}_x & 1 & 1 & 1 & 1 & 1 & -1 & -1 & 0 & 1 & 1 & -1 & -1 & -1 & -1 & -1 & 1 & 1 & 0 & -1 & -1 \\ \mathrm{MCC}_y & 0 & 0 & 0 & 0 & 0 & -1 & -1 & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{array}$$

Ugotovimo, da nikoli ne more biti par verižne kode po koordinatah x in y 0, zato lahko kodo 00 uporabimo za preklapljanje med predznaki, ki ji sledi par simbolov 0 ali 1, ki pove, v kateri smeri se je spremenil predznak. Na primer: 00 10 pomeni, da spremenimo predznak kodi x, 00 01 spremeni predznak kodi y, 00 11 pa kodama v obe smeri. Na ta način dobimo nepredznačeno verižno kodo Manhattan, katere abeceda sestoji samo iz dveh znakov, $\Sigma_{UMCC} = \{0, 1\}$. Verižno kodo UMCC za objekt s slike 8.16 vidimo v razpredelnici 8.4, pri čemer predpostavimo, da je začetni predznak v obeh smereh pozitiven.

Razpredelnica 8.4: Verižna koda UMCC za objekt s slike 8.16

MCC$_x$	1	1	1	1	1	**0**	**1**	1	1	0	**0**	**1**	1	1	**0**	**1**	1	1	1	1	1	**0**	**1**	1	1	0	**0**	**1**	1	1
MCC$_y$	0	0	0	0	0	**0**	**1**	1	1	1	**0**	**0**	1	1	**0**	**0**	0	0	0	0	0	**0**	**1**	1	1	1	**0**	**0**	1	1

Kodo lahko še nekoliko skrajšamo. V smeri, ki smo jo označili z 1 in pomeni spremembo predznaka, vemo, da bo naslednji simbol imel vrednost 1, zato ga pri zapisu lahko izpustimo. Rezultat vidimo v razpredelnici 8.5, kjer simbol ␣ označuje mesto izpuščenega bita.

Razpredelnica 8.5: Skrajšana verižna koda UMCC za objekt s slike 8.16

MCC$_x$	1	1	1	1	1	**0**	**1**	␣	1	0	**0**	**1**	␣	1	**0**	**1**	␣	1	1	1	1	**0**	**1**	␣	1	0	**0**	**1**	␣	1
MCC$_y$	0	0	0	0	0	**0**	**1**	␣	1	1	**0**	**0**	1	1	**0**	**0**	0	0	0	0	0	**0**	**1**	␣	1	1	**0**	**0**	1	1

UMCC ima kot edina verižna koda zmožnost zaznati monotone dele v opisanem objektu. Monotoni deli se nahajajo med dvema spremembama predznakov v posamezni koordinatni osi. UMCC lahko uporabimo tudi kot lomno kodo.

Razpredelnica 8.6 povzema lastnosti vseh obravnavanih verižnih kod, podrobnosti pa najdemo v [99].

Verižne kode lahko tudi učinkovito stisnemo. Kot najbolj stisljiva se je izkazala koda 3OT [53, 52, 100, 48], in sicer v kombinaciji z Burrows-Wheelerjevo transformacijo, transformacijo premik naprej in interpolativnim oziroma aritmetičnim kodiranjem, ki smo jih spoznali v prejšnjih poglavjih.

Razpredelnica 8.6: Lastnosti verižnih kod

	F8 in MC	F4	VCC	3OT	UMCC
središčna koda	D	D	N	N	D
lomna koda	N	D	D	D	D
zazna ravne črte	D	D	D	D	D
loči med vod. in navp. črtami	D	D	N	N	D
zazna diagonalne črte	D	D	D	D	D
neobčutljiva na vrtenje	N	N	D	D	D
neobčutljiva na zrcaljenje	N	N	D	D	D
omogoča vrtenje	D	D	D	N	D
omogoča zrcaljenje	D	D	D	N	D
omogoča skaliranje	D	D	N	N	D
zazna monotone dele	N	N	N	N	D

Naloge

1. Naj bo slika v ločljivosti 720×576 pikslov. Kakšnega reda mora biti Hilbertova krivulja, da jo bomo lahko preslikali v prostor Hilbertovih indeksov?
2. Napišite program, ki bo narisal Hilbertovo krivuljo za podani red.
3. Napišite program, ki bo za preslikavo med 2D koordinatami in Hilbertovimi indeksi uporabljal diagram stanj.
4. V literaturi poiščite razlago Mortonove krivulje polnjenja prostora in jo implementirajte.
5. Napišite program, ki bo rasteriziran geometrijski objekt zapisal z vsemi znanimi verižnimi kodami.
6. Napišite program, ki bo opravil zrcaljenje verižne kode glede na navpičnico.
7. Napišite program, ki bo ugotovil, ali podani verižni kodi predstavljata enak geometrijski objekt.
8. V verižnih kodah poiščite pojavljanje vzorcev s priponskim drevesom.
9. Nad verižnimi kodami uporabite transformaciji MTF in IF. Ali katera zmanjša informacijsko entropijo?

10. Verižne kode transformirajte z zaporedjem transformacij BWT in MTF. Pridobljeno zaporedje indeksov stisnite s Huffmanovim, aritmetičnim in interpolativnim kodirnikom. Kakšno razmerje stiskanja ste dosegli?

11. Verižne kode stisnite z algoritmom LZW. Primerjajte učinkovitost stiskanja, ki ste ga dosegli pri prejšnji nalogi.

Literatura

[1] Cormen, T. H., C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to algorithms. MIT Press (3. izdaja), 2009.

[2] Bowers, K. J. Accelerating a particle-in-cell simulation using a hybrid counting sort. Journal of Computational Physics 173(2), 393–411, 2001.

[3] Faujdar, N., S. Ghera. Performance evaluation of parallel count sort using GPU computing with CUDA. Indian Journal of Science and Technology 9(15), 1–12, 2016.

[4] Čuk, R. Algoritmi za urejanje – interni zapiski. UM FERI, Inštitut za računalništvo, 2005.

[5] Hollerith, H. Art of compiling statistics. Patent US395781A, 1889.

[6] Maus, A. RadixInsert, a much faster stable algorithm for sorting floating-point numbers. 32. Norsk Informatikkonferanse, NIK 2019, Bibsys Open Journal Systems, 2019.

[7] Satish, N., M. Harris, M. Garland. Designing efficient sorting algorithms for manycore GPUs. IEEE International Symposium on Parallel Distributed Processing, IEEE, 1–10, 2009.

[8] Bandyopadhyay, S., S. Sahni. GRS – GPU radix sort for multifield records. International Conference on High Performance Computing, 1–10, 2010.

[9] Burnetas, A., D. Solow, R. Agarwal. An analysis and implementation of an efficient in-place bucket sort. Acta Informatica 34, 687–700, 1997.

[10] Corwin, E., A. Logar. Sorting in linear time – variations on the bucket sort. Journal of Computing Sciences in Colleges 20(1), 197–202, 2004.

[11] Faujdar, N., S. Saraswat. The detailed experimental analysis of bucket sort. 7. International Conference on Cloud Computing, Data Science & Engineering, 1–6, 2017.

[12] Chlebus, B. S. A parallel bucket sort. Information Processing Letters 27(2), 57–61, 1988.

[13] Navarro, G. A guided tour to approximate string matching. ACM Computing Surveys 33(1), 31–88, 2001.

[14] Carras, C., T. Lecroq. Handbook of exact string-matching algorithms. Springer, 2008.

[15] Karp, R. M., M. O. Rabin. Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development 31(2), 249–260, 1987.

[16] Khan, Z. A., R. K. Pateriya. Multiple pattern string matching methodologies: A comparative analysis. International Journal of Scientific and Research Publications 2(7), 498–504, 2012.

[17] Sharma, J., M. Singh. CUDA based Rabin-Karp pattern matching for deep packet inspection on a multicore GPU. International Journal for Computer Network and Information Security 10, 70–77, 2015.

[18] Leonardo, B., S. Hansun. Text documents plagiarism detection using Rabin-Karp and Jaro-Winkler distance algorithms. Indonesian Journal of Electrical Engineering and Computer Science 5(2), 462–471, 2017.

[19] Knuth, D., J. H. Morris, V. Pratt. Fast pattern matching in strings. SIAM Journal on Computing 6(2), 323–350, 1977.

[20] Horspool, R. N. Practical fast searching in strings. Software: Practice and Experience 10(6), 501–506, 1980.

[21] Boyer, R. S., J. S. Moore. A fast string searching algorithm. Communications of the ACM, 20(10), 762–772, 1977.

[22] Sunday, D. M. A very fast substring search algorithm. Communications of the ACM 33(8), 132–142, 1990.

[23] Wagner, R., M. Fischer. The string to string correction problem. Journal of ACM 21(1), 168–178, 1974.

[24] Trappe, W., L. C. Washington. Introduction to cryptography with coding theory. Pearson Education International, 2006.

[25] Jakopin, P. Zgornja meja entropije pri leposlovnih besedilih v slovenskem jeziku. Doktorska disertacija, Univerza v Ljubljani, Fakulteta za elektrotehniko, 1999.

[26] Frekvence črk. `https://sl.wikipedia.org/wiki/Frekvence_crk`, (dostop: 2020).

[27] Shannon, C. E. A mathematical theory of communication. Bell System Technical Journal (27), 379–423, 1948.

[28] Fano, R. M. The transmission of information. Tehniško poročilo št. 65, Research Laboratory of Electronics at MIT, 1949.

[29] Huffman, D. A method for the construction of minimum–redundancy codes. Proceedings of the IRE 40(9), 1098–1101, 1952.

[30] Salomon, D., G. Motta. Handbook of data compression. Springer (5. izdaja), 2010.

[31] Sayood, K. Introduction to data compression. Morgan Kaufmann (4. izdaja), 2012.

[32] Nelson, M., J.-L. Gailly. The data compression book. M&T Books (2. izdaja), 1996.

[33] Pu, I. M. Fundamental data compression. Butterworth–Heinemann, 2006.

[34] Skibinski, P. Improving HTML compression. Informatica 33(3), 363–373, 2009.

[35] Furht, B. A survey of multimedia compression techniques and standards. Part I: JPEG standard. Real Time Imaging 1(1), 49–67, 1995.

[36] Abel, J. Post BWT stages of the Burrows-Wheeler compression algorithm. Software: Practice and Experience 40(9), 751–777, 2010.

[37] Bodden, E., M. Clasen, J. Kneis. Arithmetic coding revealed–A guided tour from theory to praxis. Tehniško poročilo št. 2007-5, McGill University, School of Computer Science, Sable Research Group, 2007.

[38] Said, A. Introduction to Arithmetic Coding – Theory and Practice. Tehniško poročilo št. HPL–2004–76, Imaging Systems Laboratory, HP Laboratories Palo Alto, 2004.

[39] Ziv, J., A. Lempel. A universal algorithm for sequential data compression. IEEE Transactions on Information Theory 23(3), 337–343, 1977.

[40] Deutsch, P. `http://www.ietf.org/rfc/rfc1951`, (dostop: 2021).

[41] Ziv, J., A. Lempel. Compression of individual sequences via variable-rate coding. IEEE Transactions on Information Theory 24(5), 530–636, 1987.

[42] Welch, T. A. High speed data compression and decompression apparatus and method. Patent US4558302B1, 1983.

[43] Golomb, S. W. Run-length encodings. IEEE Transactions on Information Theory IT–12(3), 399–401, 1966.

[44] Rice, R. F. Some practical universal noiseless coding techniques. Tehniško poročilo št. JPL-PUB-79-22, Jet Propulsion Laboratory, California Institute of Technology, 1979.

[45] Robinson, T. SHORTEN: Simple lossless and near-lossless waveform compression. Tehniško poročilo št. CUED/F-INFENG/TR. 156, Cambridge University, 1994.

[46] Liebchen, T. MPEG-4 ALS – The Standard for Lossless Audio Coding. The Journal of the Acoustical Society of Korea 28(7), 618–629, 2009.

[47] Moffat, A., V. N. Anh. Binary codes for locally homogeneous sequences. Information Processing Letters 99(5), 175–180, 2006.

[48] Žalik, B., D. Mongus, K. Rizman Žalik, D. Podgorelec, N. Lukač. Lossless chain code compression with an improved binary adaptive sequential coding of zero-runs. Journal of Visual Communication and Image Representation 75, 103050, 2021.

[49] Moffat, A., L. Stuiver. Binary interpolative coding for effective index compression. Information Retrieval 3(1), 25–47, 2000.

[50] Moffat, A., A. Turpin. Compression and coding algorithms. Kluwer Academic, 2002.

[51] Žalik, B., D. Strnad, Š. Kohek, I. Kolingerová, A. Nerat, N. Lukač, B. Lipuš, M. Žalik, D. Podgorelec. FLoCIC: A few lines of code for raster image compression. Entropy 25(3), 533, 2023.

[52] Žalik, B., K. Rizman Žalik, E. Zupančič, N. Lukač, M. Žalik, D. Mongus. Chain code compression with modified interpolative coding. Computers & Electrical Engineering (77), 27–36, 2019.

[53] Žalik, B., D. Mongus, N. Lukač, K. Rizman Žalik. Efficient chain code compression with interpolative coding. Information Sciences 439–440, 39–49, 2018.

[54] Howard, P. G., J. S. Vitter. Fast and efficient lossless image compression. Proceedings of the 1993 IEEE Data Compression Conference, 351–360, 1993.

[55] Ryabko, B. Y. Data compression by means of a 'book stack'. Problems in Information Transmission 16(4), 265-269, 1980.

[56] Žalik, B., N. Lukač. Chain code lossless compression using Move-To-Front transform and adaptive Run-Length Encoding. Signal Processing: Image Communication 29(1), 96–106, 20145.

[57] Albers, S., J. Westbrook. Self–organizing data structures. V: A. Fiat, G. J. Woeginger (ur.) Online Algorithms: The state of the art. Springer: Lecture Notes in Computer Science 1442, 31–51, 1998.

[58] Arnavut, Z., S. S. Magliveras. Block Sorting and Compression. V: J. A. Storer, M. Cohn (ur.) Proceedings of the IEEE Data Compression Conference, 181–190, 1997.

[59] Move-To-Front and inversion coding. Arnavut, Z. V: J. A. Storer, M. Cohn (ur.) Proceedings of the IEEE Data Compression Conference, 193–202, 2000.

[60] Binder, E. Distance coder. `https://groups.google.com/g/comp.compression/c/96DHNJgf0NM/m/Ep15oLxq1CcJ` (dostop: 2020).

[61] Gagie, T., G. Manzini. An experimental study of a novel Move-To-Front-or-Middle (MFM) list update algorithm. V: B. Ma, K. Zhang (ur.) Combinatorial Pattern Matching. Springer: Lecture Notes in Computer Science 4580, 71–82, 2007.

[62] Grossi, R., A. Gupta, J. Vitter. High-order entropy-compressed text indexes. Proceedings of the 14th annual ACM-SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics, 841–850, 2003.

[63] Makris, C. M. Wavelet trees: a survey. Computer Science and Information Systems 9(2), 581–625, 2012.

[64] Navarro, G. Wavelet trees for all. Journal of Discrete algorithms 25, 2–29, 2014.

[65] Burrows, M., D. J. Wheeler. A block-sorting lossless data compression algorithm. Tehniško poročilo št. 124, HP System Research Center, 1994.

[66] Adjeroh, A., A. Mukherjee, T. Bell. The Burrows-Wheeler transform: Data compression, suffix arrays, and pattern matching. Springer Science + Business Media, 2008.

[67] Mamber, M., G. Myers. Suffix arrays: a new method for on-line string searches. First Annual ACM-SIAM Symposium on Discrete Algorithms, 319–327, 1990.

[68] Nong, G., S. Zhang, W. H. Chan. Two efficient algorithms for linear time suffix array construction. IEEE Transactions on Computers 60(10), 1471–1484, 2011.

[69] Kärkkäinen, J., P. Sanders, S. Burkhardt. Linear work suffix array construction. Journal of ACM 53(6), 918–936, 2017.

[70] Spencer, C. `http://spencer-carroll.com/the-dc3-algorithm-made-simple/` (dostop: 2021).

[71] Briandais, R. De La. File searching using variable length keys. Proceedings of Western Joint Computer Conference, 295–298, 1959.

[72] Fredkin, E. Trie Memory. Communications of the ACM 3(9), 490–499, 1960.

[73] Weiner, P. Linear pattern matching algorithm. Proceedings of 14th IEEE Symposium on Switching, 1–11, 1973.

[74] McCreight, E.M. A space-economical suffix tree construction algorithm. Journal of ACM 23(2), 262–272, 1976.

[75] Ukkonen, E. On-line construction of suffix trees. Algorithmica 14(3), 249–260, 2017.

[76] Tushar, R.
`https://www.youtube.com/watch?v=aPRqocoBsFQ` (dostop: 2021).

[77] Horvat, Š. Ukkonenov algoritem konstrukcije priponskega drevesa. UM FERI, diplomsko delo univerzitetnega študija Računalništvo in informacijske tehnologije, 2020.

[78] Lawder, J. K., P. J. H. King. Using space-filling curves for multidimensional indexing. V: B. Lings, K. G. Jeffey (ur.) Proceedings of the 17th British national conference on databases: Advances in databases, Springer: Lecture Notes in Computer Science 1832, 20–35, 2000.

[79] Morton, G. M. A computer oriented geodetic data base; and a new technique in file sequencing. Tehniško poročilo: IBM, 1996.

[80] Gray, F. Pulse code communication. Patent US2632058, 1953.

[81] Sagan, H. Space-filling curves. Springer-Verlag, 1994.

[82] Lawder, J. K. The application of space-filling curves to the storage and retrieval of multi-dimensional data. Doktorska disertacija, University of London, 2000.

[83] Bader, B. Space-filling curves – An introduction with applications in scientific computing. Springer-Verlag, 2013.

[84] Chung, K.-L., Y.-L. Huang, Y.-W. Liu. Efficient algorithms for coding Hilbert curve of arbitrary-sized image and application to window query. Information Sciences 177(19), 2130–2151, 2007.

[85] Asano, T., D. Ranjan, T. Roos, E. Welzl, P. Widmayer. Space-filling curves and their use in the design of geometric data structures. Theoretical Computer Science 181(1), 3–15, 1997.

[86] Butz, A. R. Alternative algorithm for Hilbert's space-filling curve. IEEE Transactions on Computers C-20(4), 424–426, 1971.

[87] Žalik, B., D. Mongus, K. Rizman Žalik, N. Lukač. Boolean operations on rasterized shapes represented by chain codes using space filling curves. Journal of Visual Communication and Image Representation 49, 420–432, 2017.

[88] Bially, T. Space-filling curves: Their generation and their application to bandwidth reduction. IEEE Transactions on Information Theory 15(6), 658–664, 1969.

[89] Žalik, B., D. Strnad, K. Rizman Žalik, A. Nerat, N. Lukač, B. Lipuš, D. Podgorelec. Pregled 2D verižnih kod. V: A. Žemva, A. Trost (ur.) Zbornik tridesete mednarodne Elektrotehniške in računalniške konference ERK 2021, 329–332, 2021.

[90] Freeman, H. On the encoding of arbitrary geometric configurations. IRE Transactions on Electronic Computers EC10, 260–268, 1961.

[91] Freeman, H. Computer processing of line drawing images. ACM Computing Surveys 6(1), 57–97, 1974.

[92] Liu, Y. K., B. Žalik. An efficient chain code with Huffman coding. Pattern Recognition 38(4), 553–557, 2005.

[93] Bribiesca, E. A geometric structure for two-dimensional shapes and three-dimensional surfaces. Pattern Recognition 25(5), 483–496, 1992.

[94] Nunes, P., F. Pereira, F. Marqués. Multi-grid chain coding of binary shapes. V: Proceedings of the 1997 International Conference on Image Processing 3, 114–117, 1997.

[95] Bribiesca, E. A new chain code. Pattern Recognition 32(2), 235–251, 1999.

[96] Sánchez-Cruz, H., R. M. Rodríguez-Dagnino. Compressing bi-level images by means of a 3-bit chain code. SPIE Optical Engineering 44(9), 1–8, 2005.

[97] Shih, F. Y., W.-T. Wong. A new single-pass algorithm for extracting the mid-crack codes of multiple regions. Journal of Visual Communication and Image Representation 3(3), 217–224, 1992.

[98] Pjević, G. Algoritem stiskanja verižne kode MD. UM FERI, diplomsko delo univerzitetnega študija Računalništvo in informacijske tehnologije, 2021.

[99] Žalik, B., D. Mongus, Y. K. Liu, N. Lukač. Unsigned Manhattan chain code. Journal of Visual Communication and Image Representation 38, 186–194, 2016.

[100] Žalik, B., D. Mongus, K. Rizman Žalik. Information Sciences 525, 109–118, 2020.

Stvarno kazalo

3OT, 180

ADFGX, 38
ADFGXV, 38
aritmetično kodiranje, 53, 68
  razširjanje delovnega intervala, 77
  s pomikanjem, 73
  s skaliranjem, 73, 77
aritmetično kodiranje s pomikanjem, 77
ASCII, 45, 50, 63, 94

BASC, 100, 103
BASCRB, 103
Bellmanova enačba, 28
binarno drevo, 124
brezizgubno stiskanje, 44
Burrows-Wheelerjeva transformacija, 48, 128
BWT, 48, 128, 133, 134
BWT-indeks, 128

cena urejanja, 27

delovni interval, 69, 77
diferenčna vozliščna koda, 177
dinamični Huffmanov algoritem, 60
drevo valčkov, 124
drseče okno, 15, 16, 87

elementarni problem, 28
enakomerna porazdelitev, 12
entropija informacij, 48

F4, 177
F8, 176
faktor stiskanja, 44
FELICS, 103, 108
fraza, 85
Freemanova verižna koda v osem smeri, 176
Freemanova verižna koda v štiri smeri, 177

geometrijska porazdelitev, 58, 96
glava datoteke, 55, 60
Golomb-Riceovo kodiranje, 99
Golombovo kodiranje, 96, 103, 108
GZIP, 88

Hilbertova krivulja
  diagram stanj, 173
  indeks, 171
  konstrukcija, 169
  n-točka, 171
  predstavitev z drevesom, 171
hitro urejanje, 5
Horspoolov algoritem, 22
  hevristika, 23
Huffmanov algoritem, 55, 60, 66, 85
Huffmanov algoritem s prilagajanjem, 60
Huffmanova koda, 56, 58, 66
Huffmanovo kodiranje, 52, 68, 96, 178

IBWT, 128, 135
IF, 118, 123
informacijska entropija, 49, 115, 117
interpolativno kodiranje, 104
intuitivne metode stiskanja, 45
inverzna Burrows-Wheelerjeva transformacija, 128
inverzne frekvence, 118
iskanje vzorca v zaporedju, 15
  hitri iskalni algoritem, 24
  Horspoolov algoritem, 22
  Knut-Morris-Prattov algoritem, 19
  naivni pristop, 16
  preizkus, 15
  Rabin-Karpov algoritem, 16
  Sundayev algoritem, 24
izgubno stiskanje, 44

ključ, 34, 35
KMP, 19
kmpNext, 21
Knut-Morris-Prattov algoritem, 19
koda s spremenljivo dolžino, 50, 104
kodirno-dekodirno drevo, 53, 55
končna aritmetika, 72
korensko urejanje, 11, 137
kriptografija, 33
krivulja polnjenja prostora, 167
kumulativna vrednost, 7
kumulativna vsota ponovitev, 7
kvantizacija, 44

lastnost dvojčkov, 60
LCS, 164
Levenštejnova razdalja, 27
lomna verižna koda, 177
LZ77, 87, 88
LZ78, 91
LZSS, 88
LZW, 93

MCCC, 181
modificirana binarna koda, 97
Morsejeva abeceda, 38
MTF, 48, 115

naivni pristop, 16
najdaljše skupno podzaporedje, 164
najkrajša razdalja urejanja zaporedij, 27
nepopolno priponsko drevo, 150
Nepredznačena verižna koda Manhattan, 182

obhod po nivojih, 61
Ogliščna verižna koda, 179
operacije urejanja, 27

permutacija, 128
PkZip, 88
Playfair, 36
pomikalni šifrirnik, 33
posplošeno priponsko drevo, 164
predpona, 21
predponska koda, 50, 55, 66
predponske kode, 53
predstavljiva števila, 72
prefiksna koda, 50
prepletanje, 99
pretočni algoritem, 150
pretočno stiskanje, 60, 63
prihranek stiskanja, 44
prilagodljivo binarno zaporedno kodiranje, 100, 103
prilagodljivo binarno zaporedno kodiranje z vrnitvijo, 103
pripona, 21
priponski kazalec, 154
priponsko drevo, 133, 145, 146, 150, 155, 162

- naivni algoritem, 148
- Ukkonenov algoritem, 150

priponsko polje, 133
prisekana binarna koda, 97
prostorski indeks, 167

Rabin-Karpov algoritem, 16
Razdalja urejanja zaporedij
- Bellmanova enačba, 28

razdalja urejanja zaporedij
- Wagnerjev-Fischerjev algoritem, 30

razmerje stiskanja, 44, 45
razširjanje, 61
razširjanje podatkov, 43, 55
red Hilbertove krivulje, 169
Riceovo kodiranje, 99
RLE, 47, 96, 103, 128
RS, 122

samoorganizirajoča se podatkovna struktura, 117
Shannon-Fanojev algoritem, 53, 55, 57, 60, 85
- razširjanje, 55

Shannonova enačba, 49, 68
Shannonova entropija, 49, 68
skaliranje delovnega intervala, 73, 77
skaliranje E1, 78
skaliranje E2, 79
skaliranje E3, 79
slovar, 145
- dinamični, 86
- s prilagajanjem, 86
- statični, 85

slučajna spremenljivka, 49
središčna verižna koda, 176
Središčno-lomna verižna koda, 181
stabilno urejanje, 5, 9, 11
statistično stiskanje podatkov
- Huffmanov algoritem, 55
- Huffmanov algoritem s prilagajanjem, 60
- Shannon-Fanojev algoritem, 52

stiskanje podatkov, 43
- algoritem LZW, 93
- brez izgub, 44
- domenskoodvisne metode, 44
- intuitivne metode, 45, 47
- splošnonamenske metode, 44
- z izgubami, 44

stiskanje s prilagajanjem, 66
stiskanje zaporedja enakih znakov, 47
Sundayev algoritem, 24
surovi podatki, 43

transformacija desno manjše, 122
transformacija inverzne frekvence, 123
transformacija kodiranje razdalj, 121
transformacija pomik naprej, 48, 115
transformacija z drevesom valčkov, 124
trie, 145
triortogonalna verižna koda, 180

ubežna koda, 63
Ukkonenov algoritem, 150, 155
- aktivna dolžina, 155
- aktivna povezava, 155
- aktivna točka, 155
- aktivno vozlišče, 155

UMCCT, 182
unarna koda, 97
uniformna porazdelitev, 12
urejanje na mestu, 5
urejanje Roman, 10
urejanje s kopico, 5
urejanje z izbiranjem, 5

urejanje z mehurčki, 5
urejanje z vedri, 11
urejanje z vstavljanjem, 5
urejanje z zlivanjem, 5
usmerjena 8-smerna Freemanova vozliščna koda, 178

varianca kode, 60
VCC, 179
verižna koda razlik, 178
verižne kode, 167, 176
- 3OT, 180
- DCC, 178
- diferenčna vozliščna koda, 177
- F4, 177
- F8, 176
- MCCC, 181
- UMCC, 182
- usmerjena 8-smerna Freemanova vozliščna koda, 178
- VCC, 179
- verižna koda razlik, 178

Vigenérjev šifrirnik, 35
VLC, 50, 52, 53, 56, 60, 104, 115
- varianca kode, 60

vrteča se zgostitvena funkcija, 17
vzorec, 15, 162

Wagner-Fischerjev algoritem, 30
WT, 124
WTT, 124

zaporedje, 15, 27
zgostitvena funkcija, 17
zgostitvena vrednost, 17
ZLIB, 88
ZRT, 47, 48

čelno stiskanje, 46

šifriranje, 33
šifrirnik, 33, 35, 36, 38
- enočrkovni, 34
- veččrkovni, 34

številsko drevo, 93, 145
števno urejanje, 6, 10, 11
žeton, 85, 115

APLIKACIJE RAČUNALNIŠKIH ALGORITMOV

BORUT ŽALIK
Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Maribor, Slovenija
borut.zalik@um.si

Učbenik Aplikacije računalniški algoritmov je namenjen študentom prve stopnje študijskega programa računalništvo in informacijske tehnologije s ciljem, spoznati algoritme, ki jih uporabniki pri svojem delu pogosto uporabljajo. Z implementacijo teh algoritmov bodo študentje pri prepotrebno rutino za za vstop v umetnost programiranja univerzalnega stroja, to je računalnika. Učbenik prinaša naslednje vsebine: urejanje podatkov v linearnem času, iskanje vzorcev v nizih, iskanje minimalne razdalje urejanja, preproste šifrirnike, metode brezizgubnega stiskanja podatkov, metode transformacije nizov, priponska polja in priponska drevesa ter algoritme v rastrskem prostoru (verižne kode in krivulje polnjenja prostora).

Ključne besede:
urejanje podatkov v linearnem času,
iskanje vzorcev v nizih,
iskanje minimalne razdalje urejanja,
brezizgubno stiskanje podatkov,
transformacije nizov,
priponska polja in priponska drevesa,
verižne kode,
krivulje polnjenja prostora

Univerzitetna založba Univerze v Mariboru

DOI https://doi.org/10.18690/um.feri.6.2023
ISBN 978-961-286-752-2

Univerza v Mariboru

Fakulteta za elektrotehniko,
računalništvo in informatiko