Algoritmi
Domača naloga 2

Sara Bizjak | 27202020

April 2021

Problem 1 - Ujemanje vzorcev

A del:

Za podan vzorec s izračunamo KMP preponsko funkcijo π.

$$s = \texttt{ababbabbabbababbabb}$$

Podan imamo niz s dolžine n. Preponska funkcija je definirana kot seznam π dolžine n, kjer je $\pi[i]$ dolžina najdaljše ustrezne predpone podniza $s[0 \dots i]$, kar je tudi pripona tega podniza. Ustrezna predpona niza je definirana kot predpona, ki ni enaka samemu nizu. Po definiciji velja

$$\pi[0] = 0,$$

definicijo predponske funkcije pa matematično zapišemo kot

$$\pi[i] = \max_{k=0,\dots,i} \{k : s[0 \dots k-1] = s[\ i-(k-1) \dots i]\}$$

Koda za iskanje KMP preponske funkcije π:

```
def prefix_function(s):
    n = len(s)
    pi = [0 for i in range(n)]
    for i in range(1, n):
        k = pi[i - 1]
        while k > 0 and s[i] != s[k]:
            k = pi[k - 1]
        if s[i] == s[k]:
            k = k + 1
        pi[i] = k
    return pi
```

Program vrne $\pi = [0, 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 3, 4, 3, 4, 5, 6, 7, 8]$.

C del:

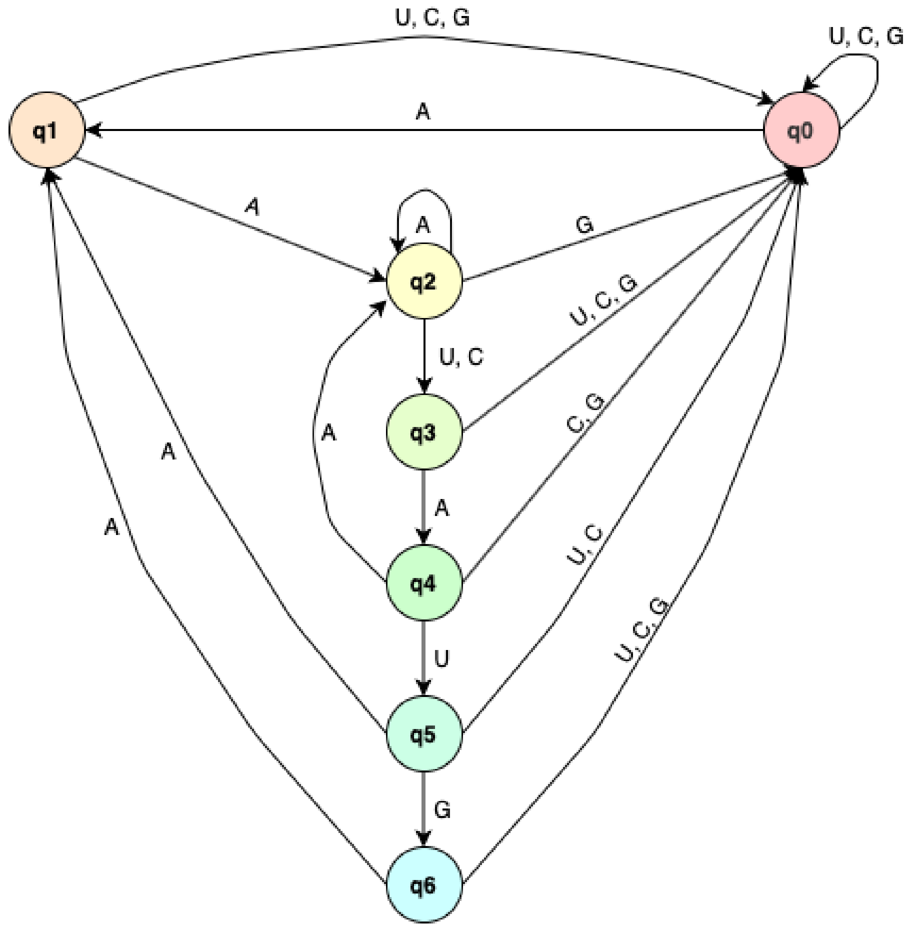
Sestavimo končni avtomat za iskanje `asparagina (AAU, AAC)` in `metionina (AUG)`. Napišemo program, ki genom prebere in s pomočjo avtomata vrne lokacija vseh pojavitev želenih amino kislin. Z drugimi besedami, iščemo končni avtomat, ki bo v genomu poiskal vse pojavitve vzorcev `AAUAUG` in `AACAUG`.

Avtomat predstavimo kot:

- množica stanj $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$
- začetno stanje q_0
- množica sprejemajočih stanj $A = \{q_6\}$
- vhodna abeceda $\Sigma = \{A, U, C, G\}$
- prehodna funkcija δ

Tabela prehodne funkcije δ:

`stanje / vhod`	`A`	`U`	`C`	`G`
q_0	q_1	q_0	q_0	q_0
q_1	q_2	q_0	q_0	q_0
q_2	q_2	q_3	q_3	q_0
q_3	q_4	q_0	q_0	q_0
q_4	q_2	q_5	q_0	q_0
q_5	q_1	q_0	q_0	q_6
q_6	q_1	q_0	q_0	q_0



Slika 1: Grafični prikaz končnega avtomata za iskanje pojavitev vzorcev `AAUAUG` in `AACAUG`.

Napišemo še program, ki prebere genom in s pomočjo zgoraj generiranega avtomata vrne lokacije vseh pojavitev želenih aminokislin, tj. vzorcev `AAUAUG` in `AACAUG`. Koda in primer sta dostopna v pythonovi datoteki `dn2_tools.py`.

Output priloženega primera iz datoteke `generated_genome.txt` je:
[(87, 92), (741, 746), (2601, 2606), (5802, 5807), (12686, 12691), (13491, 13496), (16546, 16551), (20695, 20700), (22791, 22796), (27087, 27092), (29068, 29073), (32418, 32423), (33616, 33621), (35313, 35318), (38418, 38423), (38512, 38517), (39254, 39259), (40316, 40321), (47611, 47616), (48406, 48411), (49426, 49431), (49480, 49485), (50297, 50302), (52819, 52824), (56505, 56510), (59779, 59784), (60732, 60737), (64416, 64421), (66670, 66675), (69173, 69178), (70362, 70367), (71221, 71226), (71765, 71770), (72389, 72394), (72727, 72732), (73763, 73768), (77177, 77182), (77403, 77408), (77613, 77618), (79836, 79841), (83899, 83904), (84438, 84443), (84946, 84951), (85516, 85521), (90810, 90815), (95465, 95470), (96221, 96226), (96797, 96802), (98347, 98352), (99256, 99261)] .

Problem 2 – IP posredovanje in vEB drevesa

A del:

Uporabimo Mojstrov teorem in izračunamo časovno zahtevnost operacije `vEB-Tree-Successor` vEB drevesa. Pokažemo še, da je časovna zahtevnost te operacije enaka $\mathcal{O}(\log\log M)$, kjer je M velikost univerzuma. Predpostavimo, da je M oblike 2^k.

Spomnimo se najprej Mojstrovega izreka, ki pravi

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + \mathcal{O}(n^c) = \begin{cases} \mathcal{O}(n^c)\ ;\ a < b^c \\ \mathcal{O}(n^c \log n)\ ;\ a = b^c \\ \mathcal{O}(n^{\log_b a})\ ;\ a > b^c \end{cases} \tag{1}$$

Vsi elementi, s katerimi se bomo ukvarjali, so števila v univerzalni množici $U = \{0, 1, \ldots, M - 1\}$, $|U| = M = 2^k$. Korensko vozlišče eVB drevesa T nad množico U hrani seznam `T.children` dolžine $\sqrt{M}$, kjer je `T.children[i]` poddrevo z vrednostmi $\{i \cdot \sqrt{M}, \ldots, (i+1) \cdot \sqrt{M} - 1\}$. Poleg tega drevo T hrani še vrednosti `T.min` in `T.max` ter pomožno eVB drevo `T.aux`, kjer

- `T.min` najmanjša vrednost, ki je trenutno v drevesu,
- `T.max` največja vrednost, ki je trenutno v drevesu,
- `T.children[i]` hrani (ostale) vrednosti x za $i = \lfloor \frac{x}{\sqrt{M}} \rfloor$,
- `T.aux` vsebuje vrednosti j samo takrat, ko `T.children[j]` ni prazen.

Za prazno drevo T definiramo `T.max` $= -1$ in `T.min` $= M$.

Algoritem, ki išče naslednika števila x v drevesu T poteka na naslednji način:

- Če $x <$ `T.min`, smo z iskanjem končali in naslednik za x je `T.min`.
- Če $x \geq$ `T.max`, potem naslednik za x ne obstaja in vrnemo M.
- Sicer definirajmo $i = \frac{x}{\sqrt{M}}$.
  - Če je $x <$ `T.children[i].max`, je naslednik vsebovan v `T.children[i]`.
  - Če je $x \geq$ `T.children[i].max`, je naslednik vsebovan v `T.aux`. Tako dobimo indeks j prvega poddrevesa, ki vsebuje naslednik elementa x. Naslednik x je v tem primeru `T.children[j].min`.

Algoritem v vsakem primeru najprej porabi $\mathcal{O}(1)$ in se potem lahko izvede še na poddrevesu velikosti $\sqrt{M}$, kar vzame $\mathcal{O}(\sqrt{M}) = \mathcal{O}\left(M^{\frac{1}{2}}\right)$, torej $\mathcal{O}\left(2^{\frac{k}{2}}\right)$.
Sledi

$$T(k) = T\left(\frac{k}{2}\right) + \mathcal{O}(1)$$

in po izreku 1 so konstante a, b in c enake:

$$a = 1,\ b = 2,\ c = 0 \text{ in zato velja } a = b^c,$$

torej

$$T(k) = \mathcal{O}(\log k) = \mathcal{O}(\log\log M)$$

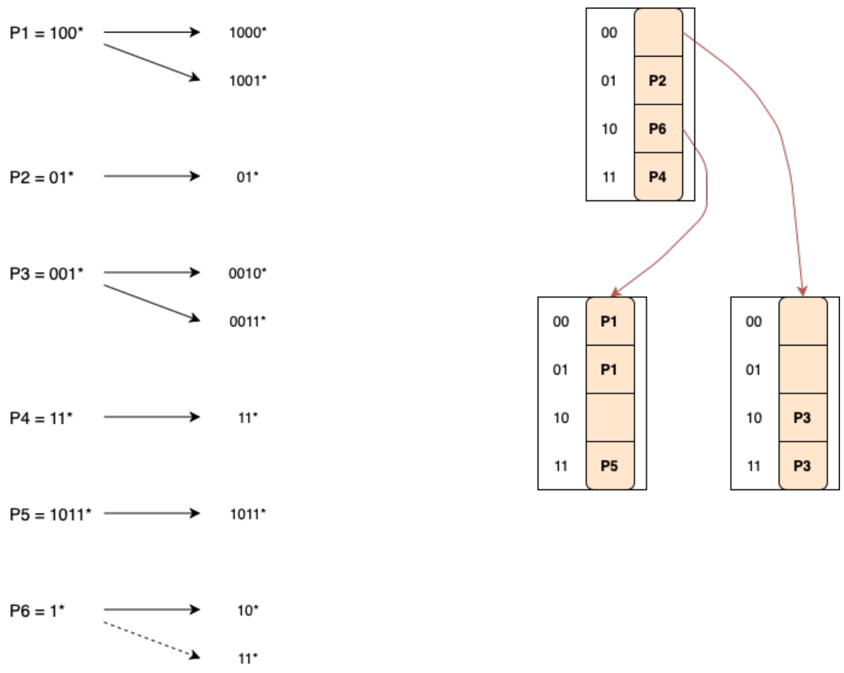
B del:

Naivni pristop pri štetju bitov s pomočjo povzetkovne tabele z Lulea algoritmom potrebuje tri pomnilniške reference. Razložimo, kako lahko prva dva stolpca združimo v enega.

V Luela algoritmu na prvih dveh pomnilniških referencah hranimo pozvetkovno tabelo P in bitno tabelo B ločeno. Tabela P je enake dolžine kot je število kosov v bitni tabeli, tabela B pa je sestavljena iz kosov določene dolžine. Označimo z M tabelo z združenim dostopom. M zgradimo tako, da P in B prepletemo tako, da pred $i-$ti kos v B shranimo $i-$ti element povzetkovne tabele. Za vsak i poznamo zgornjo mejo za $P(i)$, zato v M tej vrednosti namenimo le toliko bitov prostora, kot ga potrebuje.

Natančneje, v enem dostopu do pomnilnika je možno dobiti največ 64 bitov in ker je bitno polje dolgo največ 2^{16}, števila v tabeli P porabijo največ 16 bitov. Zato je potrebno kose v tabeli B zmanjšati na 48 bitov, da bodo skupaj s pripradajočim številom tabele P zavzeli največ 64 bitov, torej bomo lahko do elementov združene tabele M res dostopali s samo enim dostopom.

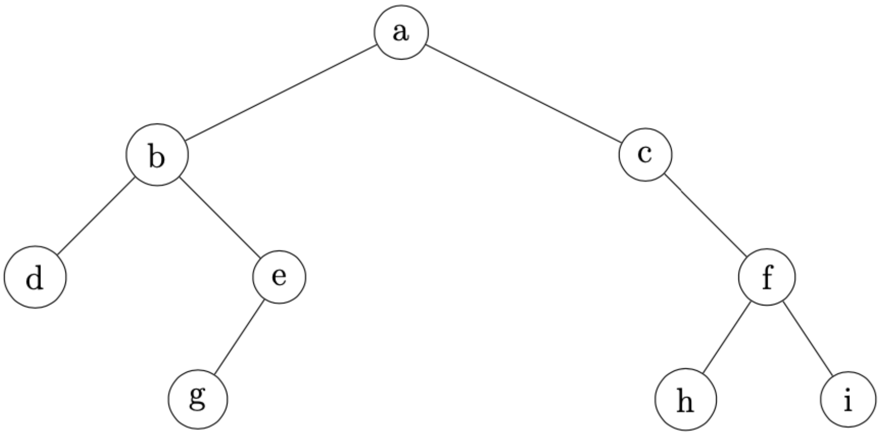
C del:



Slika 2: *Fixed stride* številsko drevo s stride velikostjo 2.

Problem 3 – Kompaktne podatkovne strukture

Podano imamo drevo.



Slika 3: Slika podanega drevesa.

A del:

Predpostavimo, da je drevo na sliki 3 ordinalno in ga zapišemo v obliki BP in LOUDS.

- `BP: ((()(()))((()())))`
- `LOUDS: 1011011010010110000`

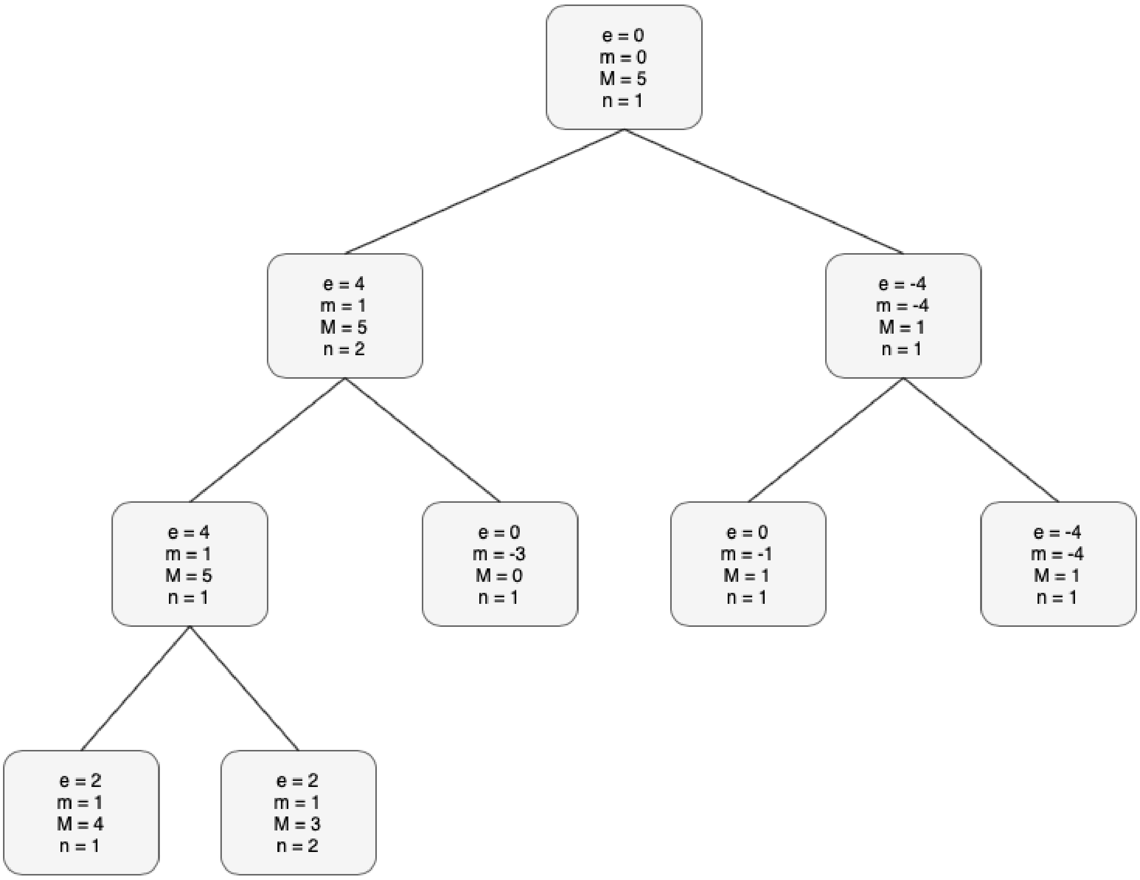
B del:

Predpostavimo, da je drevo na sliki 3 kardinalno in ga zapišimo b obliki BP in LOUDS.

- `BP: (((()())((()())()))(()((()())(()()))))`
- `LOUDS: 1011011001000010011000000000`

C del:

Za kardinalno drevo v obliki BP zgradimo `rmM-drevo` za $b = 8$.
`BP: (((()())((()())()))(()((()())(()()))))`



Slika 4: rmM-drevo za $b = 8$.

D del:

Napišemo psevdo-koda za funkcijo `Izberi(T, i)`, ki vrne i-ti element v razširjenem iskalnem dvojiškem drevesu T. Drevo T v vsakem vozlišču hrani moč levega poddrevesa $|L|$.
Funkcijo definiramo rekurzivno. V funkciji z `L` in D označimo levo in desno poddrevo, z `|L|` in `|D|` pa velikost levega in desnega poddrevesa. `T.koren` vrne koren drevesa T.

```
Izberi(T, i):
    // i-ti element je korensko vozlisce //
    ce i = |L| + 1:
        vrni T.koren
    // i-ti element je v levem poddrevesu //
    ce i < |L| + 1:
        vrni Izberi(L, i)
    // i-ti element je v desnem poddrevesu //
    sicer:
        vrni Izberi(D, i - |L| - 1)
```

Literatura

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introcustion to Algorithms*, third edition, [ogled 1. 4. 2021], dostopno na `edutechlearners.com/download/Introduction_to_algorithms-3rd%20Edition.pdf`.