3. domača naloga: Varnostno kodiranje

Električne ali magnetne motnje v računalniškem sistemu lahko povzročijo, da posamezni biti v glavnem pomnilniku (RAM) spontano spremenijo vrednost, kar lahko privede do nepravilnega delovanja računalniškega sistema. Z večanjem gostote zapisa na pomnilniških čipih in nižanjem napajalnih napetostih postaja ta težava vedno bolj pereča. Zagotavljanje visoke odpornosti pomnilniških vezij na motnje iz okolja in preprečevanje napak pri delovanju je še posebej pomembno v sistemih kot so podatkovni strežniki, industrijski krmilniki, sateliti in vesoljske sonde. Integriteta podatkov se v teh sistemih običajno zagotavlja z uporabo naprednih materialov pri gradnji, ki ščitijo podatke pred elektromagnetnim sevanjem in uporabo kodov za odkrivanje in odpravljanje napak.

Razširjeni Hammingov kod

Spekter kodov, ki se uporabljajo pri odpravljanju napak na podatkovnih medijih in v komunikaciji, je zelo širok. Zelo priljubljeni so na primer Reed-Solomonovi in konvolucijski kodi, vendar je izbor optimalnega koda za določeno aplikacijo močno odvisen od okolja, v katerem bo naprava delovala, računskih zmogljivosti strojne opreme, cene in drugih dejavnikov. Danes se na primer zaradi zagotavljanja hitrosti delovanja v pomnilnikih ECC DRAM (ang. Error Checking and Correcting Dynamic Random Access Memory) še vedno pogosto uporablja Hammingov kod H(127, 120), ki je zaradi strukture pomnilnika prilagojen tako, da z 8 varnostnimi biti ščiti 64 podatkovnih bitov. Obstajajo tudi naprednejše tehnologije za odkrivanje in odpravljanje napak, kot je IBM Chipkill, HP Chip spare in Intel SDDC.

Hammingov kod ste podrobno spoznali že na predavanjih (glej tudi [2]). Omogoča popravljanje enojnih napak, vendar ne zna razločiti med enojnimi ali večkratnimi napakami. To pomankljivost do neke mere odpravljaja razširjeni Hammingov kod oziroma SEC-DED (angl. Single error correction, double error detection), ki je v osnovi enak Hammingovemu kodu le, da ima dodan še en varnostni bit, ki je nastavljen tako, da je število enic v kodni zamenjavi vedno sodo (soda pariteta). Ta kod omogoča popravljanje enojnih napak in odkrivanje dvojnih napak. Pri tokratni domači nalogi bomo uporabili družino sistematičnih razširjenih Hammingovih kodov SEC-DED(n, k). Pri dekodiranju s SEC-DED kodom se odločamo glede na spodnjo tabelo kako dekodirati kodno zamenjavo:

Pariteta	**Sindrom**	**Najverjetnejši dogodek med prenosom**	**Akcija dekodirnika**
0	0	Ni bilo napak.	Izlušči podatkovne bite.
0	$\neq 0$	Napaka na dveh bitih.	Izlušči podatkovne bite.
1	0	Napaka na paritetnem bitu.	Izlušči podatkovne bite.
1	$\neq 0$	Napaka na enem bitu.	Popravi napako s Hammingovim kodom in izlušči podatkovne bite.

CRC

CRC (ang. Cyclic Redundancy Check) je družina cikličnih kodov za odkrivanje napak, ki se uporablja predvsem v digitalnih komunikacijskih napravah za detekcijo napak pri prenosu. Zelo razširjeni so postali zato, ker jih je mogoče enostavno implementirati v strojni opremi in se še posebej dobro obnesejo pri odkrivanju napak, povzročenih zaradi šuma v komunikacijskih kanalih. Za popravljanje napak niso najbolj primerni, zato v primeru napake oddajnik sporočilo običajno pošlje ponovno. Ciklični kod CRC-32 najdemo na primer v standardih Ethernet, SATA in MPEG-2, ciklični kod CRC-16 pri komunikaciji Bluetooth in v pomnilniških karticah SD, CRC-8-CCITT pa v vgrajenih sistemih.

V tej nalogi boste implementirali kod CRC na osnovi standarda CRC-16-CCITT[1], ki se uporablja pri prenosu podatkov preko Bluetooth. Parametri standarda so naslednji:

- Polinom: 0x1021 $\rightarrow g(p) = p^{16} + p^{12} + p^5 + 1$,
- Začetna vrednost registra: `0xFFFF`,
- Primer vrednosti CRC za niz `'123456789'`: `0x29B1`. Vsak znak vhodnega niza je v tem primeru kodiran z 8 biti (razširjen ASCII), CRC pa je prikazan šestnajstiško.

Delovanje vašega programa lahko preverite tudi preko spletne strani `http://crccalc.com/`. V tem primeru glejte rezultate v vrstici `CRC-16/CCITT-FALSE`.

Naloga

V tokratni nalogi boste spoznali uporabo linearnih bločnih kodov za simetrični binarni komunikacijski kanal [1]. Scenarij gre takole:

1. Generirali smo binarni niz (sporočilo) z.
2. Preden sporočilo z pošljemo v komunikacijski kanal, ga obogatimo z m varnostnimi biti. V kanalu namreč lahko pride do napak. Za varovanje smo uporabili enega izmed sistematičnih razširjenih Hammingovih kodov SEC-DED(n, k), kjer je $n = 2^m$ in $k = n - m$. Tako zavarovano sporočilo z sedaj imenujemo x.
3. Zavarovano sporočilo x pošljemo po kanalu. Med prenašanjem se lahko njegova vsebina pokvari, kar pomeni, da se določenemu številu bitov obrne vrednost.
4. Na izhodu iz kanala sporočilo prevzameš ti. Tvoja naloga je, da poskusiš popraviti morebitne enojne napake v njem. Pri tem seveda uporabiš ustrezen kod. Sporočilu, ki pride iz kanala, pravimo y. Ko odkodiraš y, dobiš $\hat{z}$, ki je v najboljšem primeru enak poslanemu sporočilu z.
5. Sporočilo $\hat{z}$ vrneš kot rezultat v vrstičnem binarnem vektorju `izhod`.

[1] `https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=441541`, stran 146.

6. Nad y izračunaš vrednost CRC po zgoraj določenem standardu in jo vrneš kot rezultat v šestnajstiškem zapisu v spremenljivki `crc`. Šestnajstiško število naj bo zapisano kot niz štirih znakov, npr. `'00FF'`.

Napišite funkcijo z imenom `naloga3` v programskem jeziku Python. Funkcija mora implementirati dekodiranje sporočil y, ki ste jih prejeli iz zašumljenega kanala. Funkcija `naloga3` kot vhodne argumente sprejme seznam bitov `vhod` in celo število `n` – dolžino kodne zamenjave. Argument `n` definira razširjeni Hammingov kod, ki ga je potrebno uporabiti. Seznam `vhod` predstavlja sporočilo y na izhodu iz kanala, njegovi elementi so ničle in enice (tip `int`). Vrnjena vrednost funkcije je terka: prvi element terke je seznam `izhod`, ki predstavlja odkodirano sporočilo, sestavljeno iz podatkovnih bitov $\hat{z}$; drugi element terke je niz `crc`, ki predstavlja vrednost CRC izračunano nad seznamom bitov `vhod`. Niz `crc` predstavlja število v šestnajstiški obliki.

Prototip funkcije:

```
def naloga3(vhod: list, n: int) -> tuple[list, str]:
    """
    Izvedemo dekodiranje binarnega niza ‘vhod‘, zakodiranega
    z razsirjenim Hammingovim kodom dolzine ‘n‘ in poslanega po
   zasumljenem kanalu.
    Nad ‘vhod‘ izracunamo vrednost ‘crc‘ po standardu CRC-16-CCITT.

    Parameters
    ----------
    vhod : list
        Sporocilo y, predstavljeno kot seznam bitov (stevil tipa int)
    n : int
        Stevilo bitov v kodni zamenjavi

    Returns
    -------
    (izhod, crc) : tuple[list, str]
        izhod : list
            Odkodirano sporocilo y (seznam bitov - stevil tipa int)
        crc : str
            Vrednost CRC, izracunana nad ‘vhod‘. Niz stirih znakov.
    """
    izhod = []
    crc = ''
    return (izhod, crc)
```

Testni primeri

Na učilnici se nahaja arhiv `tis-naloga3.zip`, ki vsebuje tri testne primere v obliki datotek `JSON`. Priloženo imate tudi funkcijo `test_naloga3`, ki jo lahko uporabite za preverjanje pravilnosti vaše rešitve. Pri testiranju vaše funkcije upoštevajte naslednje:

- v vhodnem seznamu bo vedno manj kot 500.000 bitov,
- najdaljša možna dolžina kodne zamenjave je 256 bitov,
- verjetnost za napako na posameznem bitu $p_e \leq 0{,}5$,
- pri posameznem testnem primeru dobite pol točke, če se `izhod` ujema z rešitvijo,
- drugo polovico točke dobite, če se tudi `crc` ujema z rešitvijo,
- izvajanje funkcije je časovno omejeno na 15 sekund,
- dovoljena je uporaba standardne knjižnice Python 3.11 (`https://docs.python.org/3.11/library/`) in paketov `numpy` (priporočamo) ter `scipy`.

Literatura

[1] D.G. Luenberger: Information Science, Princeton University, pogl. 6, 2006.

[2] H. S. Warren: Hacker's Delight, Second Edition, Addison-Wesley, Boston, 2012.