## שאלה 1

ג. מבנה הנתונים המתאים יהיה מילון כי הוא יאפשר לנו להשוות לממצאי העבר ב$O(1)$ פחות או יותר (בגלל השימוש בhush).

ד. החיסרון העיקרי של הפונקציה fingerprint הוא שהיא משתמשת בסכום ערכי הפיקסלים כדי להעניק למטריצה טביעת אצבע אריתמטית ייחודית. זה יוצר מצב בהם למטריצות שונות יש אותה טביעת אצבע (כלומר, הפונקציה אינה חח"ע).
דרך לפתור את הבעיה הזאת היא להשתמש בשיטת הורנר כדי לחשב טביעת אצבע מתאימה, ולהפעיל עליה מודולו של מספר ראשוני גדול כלשהו.

## שאלה 2

א) נוכיח באינדוקציה:
נבדוק עבור n = 4: a4<a1+a2+a3<5 (?)
3<1+1+2<5 כלומר מתקיים.

נניח נכונות הטענה עבור כל מספר עד n-1: $a(n-2)<a(n-1)<\sum_{i=1}^{n-2}(a_i)<an$

נבדוק עבור n: $\sum_{i=1}^{n-1}(a_i) = \sum_{i=1}^{n-2}(a_i) + a(n-1)$

$$an = a(n-1) + a(n-2) < \sum_{i=1}^{n-2}(a_i) + a(n-1) < an + a(n-1) = a(n+1)$$

כלומר: $an< \sum_{i=1}^{n-1}(a_i)< a(n+1)$ **מ.ש.ל.**

ב) אורך קידוד האפמן הקצר ביותר יהיה **1**, והארוך ביותר יהיה **n-1**. זה בגלל שכפי שהוכחנו, סכום כל האיברים הקטנים מאיבר בסדרה, גדול ממנו, אך קטן מהאיבר שבא אחריו. ולפי איך שקידוד האפמן עובד, נקבל שכל פעם סכום האיברים יתחבר עם האיבר הבא בסדרה, עד לאיבר האחרון, שיהיה מחובר ישירות לשורש העץ.

ג) נתונים: a1<a2<a3<….<ak, **וכן** aK<2*a1. K = 128

מהנתונים נובע ש $a1 + a1< a2 + a1$ **<-------------** $a3 < a4 < …. < ak < 2*a1< a1 + a2$

לפי קידוד האפמן, בכל שלב תמיד סוכמים את האיברים הקטנים ביותר באותו שלב בסדרה.
לכן, לפי מה שהוכחנו, בשלב הבא המספרים הקטנים ביותר יהיו a3, a4
קל לראות שבשלב הבא: a5<a6<….<ak<2*a1<a1+a2<a3+a4.

ניתן לראות שלפי קידוד האפמן, בסופו של דבר נקבל עץ מאוזן למדי (במקרה זה לחלוטין).
בגלל ש –k = 128 = 2**7, נוכל להסיק שכל איבר יהיה בן 7 ביטים.

כלומר, $|C(p)| = 7$ **,ו-** $|C(q)| = 7$ עבור a1 **ו-**ak בהתאמה.

נחשב את ההפרש: $|C(p)| - |C(q)| = 7 – 7 = 0$ **מ.ש.ל**

<u>שאלה 3</u>

א. התשובה היא **שכן, לעיתים ישתלם לנו להמתין ולא לדחוס את המחרוזת.**

דוגמא למחרוזת כזאת (עבור L = 3 מול L = 4) תהייה למשל המחרוזת: **<u>"abcbcdeabcdebbcde"</u>**

דחיסה עבור L = 3 תיראה כך: **[["a","b","c","b","c","d","e",[7,3],"d","e","b",[5,4]].**
עבור כל אות יהיו לנו שמונה ביטים, עבור כל תת רשימה 18 ביטים (ע"פ הגדרות LZ).
בדחיסה זו 10 אותיות ו2 תת רשימות, סכ"ה **116 bits** = 10*8 + 2*18.

דחיסה עבור L = 4 תיראה כך: **[["a","b","c","b","c","d","e","a",[5,4],"b",[5,4]].**
בדחיסה זו 9 אותיות ו2 תתי רשימות, סכ"ה **108 bits** = 9*8 + 2*18

הנה מחרוזת שבה שכדאי להתאפק ולא לדחוס את השרשרת מיד.

ב. (1) **<u>הערכה:</u>** האפמן צריך להיות יותר יעיל, כי סביר להניח שאם יהיו חזרות – הן תהיינה קצרות מאוד (באופן הסתברותי יש 50% שתופיע האות a, ו50% סיכוי לכל אות אחרת).
בלמפל – זיו כל אות תשתרע על גבי שמונה ביטים. במקרה הזה, כל אות בקוד האפמן תשתרע על גבי חמישה ביטים בערך, או במקרה של a – ביט יחיד.
כיוון שלא צפויות חזרות, קוד LZ יאבד את היתרון שלו, ויהיה יותר חסכוני לדחוס באמצעות האפמן.

דוגמת הרצה:

```
>>> s = genString(100000)
>>> compression_ratio(s,s)
0.47823571428571426
>>> lz_ratio(s)
0.7379542857142857
>>> s = genString(100000)
>>> compression_ratio(s,s)
0.47846
>>> lz_ratio(s)
0.73892
```

למרבה השמחה, ניתן לראות שצדקתי (יחס הדחיסה של האפמן נמצא תחת compression_ratio).

(2) במקרה זה אני מעריך שקוד LZ יהיה יעיל יותר, מכיוון שההסתברות לקבל את a לעומת כל אות אחרת היא עצומה. בקוד רנדומלי יהיו לנו מחרוזות ארוכות של a, דבר שקוד LZ דוחס בשמחה. עץ האפמן לעומת זאת לא ישתנה, ולכן למרות שa שווה רק ביט אחד, כל דחיסה של מחרוזת ארוכה מ18 כבר תהייה חסכונית יותר בLZ.

דוגמת הרצה:

```
>>> s = genString(100000)
>>> compression_ratio(s,s)
0.14964714285714287
>>> lz_ratio(s)
0.08985428571428572
>>> s = genString(100000)
>>> compression_ratio(s,s)
0.14952
>>> lz_ratio(s)
0.08988857142857143
```

התוצאות תומכות בתשובה. ניתן להבין שככל שנעלה את הסתברות ההופעה של a, יעילות הדחיסה של LZ תגבר.

ג. <u>**עבור n:**</u> כנראה שייצוג הביניים של הדחיסה ייראה כך: **[0,1,[2,n-2]]**,
ויחס הדחיסה יהיה: **(2*8 + 27 +len(bin(n-1)))\7*n**
במונחים של O(...), הדחיסה היא **O(log(n)/n)**

(את הנוסחה ליחס הדחיסה מצאתי ע"י ניתוח הקוד והרצות).

דוגמא לבדיקה:
(בכל שלב מודפסים בסדר הבא: ייצוג הביניים של הדחיסה, אורך הייצוג בביטים, יחס הדחיסה)

```
>>> lz_ratio(num,w=2**12-1,max_length= 2**5-1)
['0', '1', [2, 31], [2, 31], [2, 31], [2, 31], [2, 31], [2, 31], [2, 31], [2, 31
], [2, 31], [2, 31], [2, 31], [2, 31], [2, 31], [2, 31], [2, 31], [2, 31], [2, 3
1], [2, 31], [2, 31], [2, 31], [2, 31], [2, 31], [2, 31], [2, 31], [2, 31], [2,
31], [2, 31], [2, 31], [2, 31], [2, 31], [2, 31], [2, 31], [2, 6]]
610
0.08714285714285715
>>> lz_ratio(num,w=2**12-1,max_length= len(num) -1)
['0', '1', [2, 998]]
39
0.005571428571428572
>>> num = gen01(10000)
>>> lz_ratio(num,w=2**12-1,max_length= len(num) -1)
['0', '1', [2, 9998]]
43
0.0006142857142857142
>>> num = gen01(100000)
>>> lz_ratio(num,w=2**12-1,max_length= len(num) -1)
['0', '1', [2, 99998]]
46
6.571428571428571e-05
```

שאלה 4

א)

| (x1,x2,x3) | (x1, x2, x3, x1+ x2, x1+ x3, x2+ x3, x1+ x2+x3) |
|---|---|
| (0 , 0 , 0) | (0,0,0,0,0,0,0) |
| (0 , 0 , 1) | (0,0,1,0,1,1,1) |
| (0 , 1 , 1) | (0,1,1,1,1,0,0) |
| (1 , 1 , 1) | (1,1,1,0,0,0,1) |

ב) המרחק המינימלי של הקוד היא d = 4, וזה הגיוני, כי בקידוד כל ביט משפיע בארבע מקומות שונים, לכן שינוי של ביט מהשלושה גם ישפיע בארבע מקומות שונים. הקידוד גם בנוי באופן כזה שבכל מילת קוד יופיעו ארבע אחדו בלבד, וכיוון שכל שינוי במילה המקורית מיד משפיע על ארבע ביטים, לא ייתכן (לפחות לא מצאתי) שינוי של פחות מארבע ביטים.

דוגמא לשתי מילות קוד במרחק 4: (0 , 0 , 1) ------------> (0,0,1,0,1,1,1)
(0 , 1 , 1) --------------> (0,1,1,1,1,0,0)

ג) התשובה היא שכן – קיימת מילה כזו, והיא למעשה כל אחת ממילות הקוד החוקיות. זה נובע מההסבר בסעיף ב' – כל מילת קוד חוקית תיבדל מחברתה בארבע ביטים בדיוק.

לדוגמא: (0 , 1 , 1) ----------------> (0,1,1,1,1,0,0)
(1 , 1 , 1)-----------------> (1,1,1,0,0,0,1)
(1 , 1 , 0) ----------------> (1,1,0,0,1,1,0)
(1 , 1 , 1)-----------------> (1,1,1,0,0,0,1)
(1 , 1 , 0) ----------------> (1,1,0,0,1,1,0)
(0 , 1 , 1) ----------------> (0,1,1,1,1,0,0)

כאמור עבור כל מילת קוד המרחק יהיה 4. Y יכולה להיות כל אחת מהן.

**חלק 2:**

bad_coding הוא קוד מטיפוס [n = (|x|+1)*4, k = |x|, d = 4\8]. המרחק המינימלי יהיה 4 עבור x>2, ויהיה שמונה עבור x = 2.