Q1

סעיף ב':

האלגוריתם בנוי מ-2 לולאות for שעוברות על אורך הרשימה: אחת שרצה קדימה, ואחת שרצה ב- reverse . בתוך כל לולאת for ישנה לולאת while ובסה"כ **מספר האיטרציות** בכל מקרה יהיה O(n**2+n**2)=O(n**2) . (מגיעים לכך מסכום סדרה חשבונית). לכל אינדקס ברשימה נבצע השוואה בין הרישא שלו לבין הסיפא של כל האינדקסים האחרים. (לפי הלולאה הראשונה בודקים בסדר עולה, ולפי הלולאה השנייה בסדר יורד של אינדקסים). המקרה "הגרוע" יתקבל כאשר בהשוואת הרישא והסיפא מגיעים לתוו האחרון, ואז יתקבל O(k) , ובנוסף הדבר יחול על כל האינדקסים שברשימה, זה יקרה בעצם כאשר כל המחרוזות תהיינה שוות זו לזו. יצויין כי אין זה משנה מבחינת הסיבוכיות אם התוו האחרון שווה/שונה מאחר שהוספת איבר לרשימה הסופית למשל תעלה לנו O(1) . אם כן, במקרה "הגרוע" על כל i המקיים n>i>0 נבצע O(k) **פעולות** . אם נחשב לפי סכום סדרה חשבונית נקבל שהסיבוכיות הכוללת תהיה: $O((nk-1+k)n/2)+O((nk-1+k)n/2)=O(n^2k+nk-n)=O(n^2k)$

סעיף ה':

פעולת ה-insert : O(nk) . על כל slice יהיה לנו O(k) וגם O(k) על חישוב ה-Hash . ככה נעשה עם רשימה באורך n ולכן מפעולת הכנסת הרישות למילון נשלם O(nk).

בדיקת הסיפות: משום שמניחים בסעיף זה שפונקציית prefix_sufix_overlap_hash1 תחזיר רשימה ריקה, הרי שלולאת ה- for הפנימית (שבקוד) שפועלת על רשימה שגודלה כמספר ההתאמות בין רישות לסיפות לא תבצע פעולות. משמע נשלם את אותה סיבוכיות כמו ב-insert , שכן גם כאן נעשה slice ב-O(k) ונעבור על כל הרשימה.

ובסה"כ נקבל שהסיבוכיות הכוללת תעמוד על O(nk)+O(nk)=O(nk)

Q2

תוצאות קירוב pi באמצעות 3 הגנרטורים השונים:

להלן הבדיקה שבוצעה:

```
g_monte_carlo=pi_approx_monte_carlo()
pi_monte_carlo= [ next(g_monte_carlo) for i in range(20)][19]
print("pi_monte_carlo:" , pi_monte_carlo, "### proximity to pi:",
abs(math.pi-pi_monte_carlo))

g_leibniz=pi_approx_leibniz()
pi_leibniz= [ next(g_leibniz) for i in range(20)][19]
print("pi_leibniz:" , pi_leibniz, "### proximity to pi:", abs(math.pi-
pi_leibniz))

g_integral=pi_approx_integral()
pi_integral= [ next(g_integral) for i in range(20)][19]
print("pi_integral:", pi_integral, "### proximity to pi:", abs(math.pi-
pi_integral))
```

להלן תוצאות הבדיקה:

pi_monte_carlo: 3.1417228142955915 ### proximity to pi: 0.00013016070579841

pi_leibniz: 3.141592176747568 ### proximity to pi: 4.768422363632396e-07

pi_integral: 3.14159264482808 ### proximity to pi: 8.761713132798832e-09

כפי שניתן לראות השיטה שנותנת את הקירוב הטוב ביותר ל-pi היא שיטת האינטגרל.

הקירוב השני הטוב ביותר מבין 3 השיטות המוצעות היא שיטת לייבניץ.

ושיטת מונטה-קרלו (דגימות רנדומיות) נותנת קירוב הכי פחות טוב מבין ה-3.

Q3

סעיף א':

$\{'a' : 0000000\ ,\ 'b' : 0000001\ ,\ 'c' : 000001\ ,\ 'd' : 00001\ ,\ 'e' : 0001\ ,\ 'f' : 001\ ,\ 'g' : 01\ ,\ 'h' : 1\ \}$

סעיף ב':

$a_1=0^{n-1}$, $a_2=0^{n-2}1$, $a_3=0^{n-3}1$

ובעצם לכל $i>1$: $a_i=0^{n-i}1$

הדבר נובע מהגדרת האלגוריתם של עץ האפמן שייצר את צורת השרוך אם התדירויות נקבעות ע"פ פיבונאצ'י. כי לפי פיבונאצ'י:

$a_n > \sum_{i=1}^{n-2} a_i$ and $a_n>a_{n-1}$ (שוויון אם $\sum_{i=1}^{n-2} a_i + 1 = a_n$)

ולכן הצומת הפנימי שלאחריו יהיה מורכב מ-a_{n-2} ומ-a_{n-1} ומכאן תיווצר צורת השרוך של עץ האפמן. כלומר בכל שלב בבניית העץ נמזג את האיבר הבא בסדרה עם מה שמוזג עד כה ולכן נקבל עץ בעומק מקסימלי שהוא n-1 .

סעיף ג':

נתחיל לקבץ צמתים פנימיים לעץ האפמן. נבחר כמובן את a_1 ואת a_2 . מכיוון שמתקיים

$2a_1>a_n>...>a_1$ כעת המספרים הקטנים ביותר יהיו a_3 , a_4 . זה כי כמובן $a_5...a_n$ גדולים מהם וכאמור אם נסמן $a_2=a_1+k$ עבור k טבעי כלשהו הרי ש : $a_1+a_2=2a_1+k>a_4>a_3$.

באופן זה נבחר 128 צמתים פנימיים. כלומר בוחרים בצמדים את האיברים כך שבכל פעם בוחרים את התווים להיות **העלים** בעץ האפמן. למעשה במצב זה כל העלים נמצאים בתחתית העץ משום שכל

החיבורים הם של צמתים פנימיים שלא מהווים תוו a_i עבור $1\le i\le n$. המבנה של עץ האפמן שיווצר לנו יהיה בצורה של עץ בינארי מלא (256 הוא חזקה של 2) ולכן **המרחק בין a_n ל-a_1 יהיה 0.**

סעיף ד':

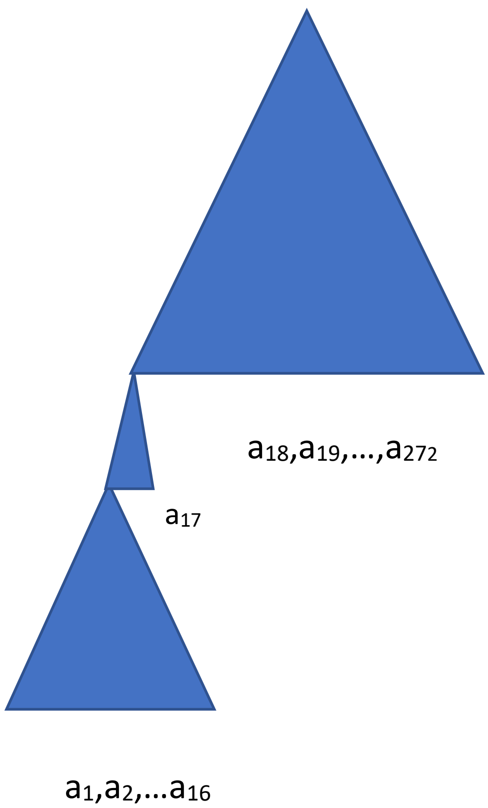
צריך לשים לב לשיקולים של חזקות של 2 כי הדבר משפיע על רמות מלאות בעץ: מאחר ש-n גדול מ-256 אך קטן מ 512, ובפרט אינו חזקה שלמה של 2 , יתקבל עץ שבו רמה אחת לא מלאה, ועל כן **ההפרש יהיה 1 במקרה זה.**

סעיף ה':

- **a** : זה בשביל שיווצרו רק צמתים פנימיים "במקביל" ויחוברו (אסיפה בצמדים). ממש כמו בסעיף ג' , יצירת עץ מלא.
- **c :** זה על-מנת שייערם עץ בינארי מאוזן עד ל- a_{16} . כלומר חיבור של צמתים פנימיים עד לאיבר ה-16 , לפני כל שאר האיברים.
- **b :** זה כדי שהמבנה יחזור על עצמו כמו בסעיפים הקודמים כך שבעצם תתחיל הספירה מ-17 ל-272 (256 צמתים).

a_{272} יקבל ייצוג של באורך 8 ביטים, ו-a_1 יהנה מייצוג באורך 13 ביטים. **ההפרש יהיה 5.**

עץ האפמן להמחשה:



Q4

סעיף א':

עבור "s="aaaa הפלט יהיה זהה בשתי הפונקציות: [['a',[1,3

סעיף ב':

אכן קיימת מחרוזת כזו: "aaakaaaaa"

ייצוג ביניים של האלגוריתם החדש: [['a','a','a','k','a',[1,4

ייצוג של המקורי: ["a","a","a","k",[4,3],"a","a"]

על ידי חישוב פשוט ניתן להבחין שהאלגוריתם החדש דוחס טוב יותר. אם נקזז מופעים חופפים נישאר עם ייצוג ביניים של המקורי [4,3] , "a" . לעומת [1,4] של החדש. כמות הביטים הנדרשת לשתי תתי-הרשימות שווה. ולמעשה החדש קצר יותר ב-8 ביטים בגלל התוו "a" .

סעיף ג':

סבורני שאין מחרוזת כזו. הדבר נעוץ בחמדנות של האלגוריתם המקורי, במובן שהוא לא מספיק "מחכה" שמא ימצא חזרה גדולה יותר. כאשר הוא ימצא חזרה הוא ימקם אותה וימשיך למצוא אולי חזרות נוספות. על אף שאולי היה יכול למצוא חזרה גדולה יותר בהמשך.

האלגוריתם החדש לעומת זאת בודק את כלל האפשרויות ובוחר בחזרה המיטיבה ביותר עם אורך הקידוד. עקרונית שורות הקוד שמבדילות בין האלגוריתם המקורי לבין החדש הן:

```
if res2_len < res1_len:
        return res2
return res1
```

על-פי שורות אלה, האלגוריתם החדש בוחר את הערך הקצר יותר מבין 2 האופציות המוצעות, לעומת המקורי שתמיד היה מחזיר את res2 . ולכן החדש ידחוס את המחרוזת טוב לפחות כמו המקורי.

Q5

סעיף א'- הקוד:

```
### Q5- generator for EC code ###

def dist(code_gen,n):
    gen_group= code_gen
    group_size= sum(1 for x in gen_group)     #an integer type

    g1= code_gen                        #going to be the main generator,
"envelope" all other generators
    min=2**(group_size)             #just for initialization
    j=2                         #index from which starts every new check,
notice: i+1=j in every iteration

    for i in range(group_size):  #the fun starts right here!

        char1= next(g1)
        g=code_gen
        for p in range(j-1):        #initializing the generator from j
            next(g)

        for k in range(j,group_size):  #generator goes to the final
char
            char_g=next(g)
            cnt=0
            for l in range(len(char1)):        #equivalent to n
                if char1[l] != char_g[l]:      #checking the distance
                    cnt+=1
            if min>cnt:
                min=cnt
        j+=1

    return min
```

<u>סעיף ב'- ניתוח הסיבוכיות:</u>

בהתחלה מריצים לולאת for בגודל 2^k איברים. כל איבר באורך n ולכן נקבל O(n*2^k) . אך החסם הנ"ל הולך להיבלע בין כה וכה בבדיקת המרחקים שעושים בהמשך.

מספר הבדיקות שנעשה על-מנת למצוא את מרחק המינג המינימלי הוא סכום של סדרה חשבונית: $\sum_{i=1}^{2**(k-1)} i$ =2^{2k-1}, שכן נצטרך לעבור על כל $x \neq y \in Imc$. בנוסף, על כל בדיקה של n כמות הביטים נידרש ל- O(n) . כלומר בסה"כ <u>**$O(n*2^{2k})=O(n*4^k)$ וזהו החסם ההדוק**</u>.

<u>**קיבלנו שסיבוכיות הזמן אקספוננציאלית ב-k ופולינומיאלית ב-n .**</u>

<u>סעיף ג':</u>

השוויון נכון.

ראינו בהרצאות ש: $(d-1)/2$ ערך תחתון מייצג את מספר השגיאות שניתנות ל**<u>תיקון</u>**.

$d-1$ לעומת זאת הוא מספר השגיאות שניתנות ל**<u>זיהוי</u>**. לפי אחת מההגדרות מההצאות:

$|B(z,d-1) \cap ImC|=1$ כי הקבוצה מכילה את z בלבד. כך שאם אפילו נקטין עוד יותר את הרדיוס נקבל כדור שמכיל את z בלבד.

<u>סעיף ד':</u>

הא"ש אינו נכון.

דוגמה נגדית תהיה עם פרמטר חזרה $t=2$. כאשר $n=6,\ k=2,\ d=2$ כלומר עבור $(6,2,2)$ קוד לא טריוויאלי.

עבור $z=$"10" ← "1100" . מתקיים $1>(d-1)/2$, $|B(z,1) \cap ImC|=1$ שאינו מכבד את הא"ש בתרגיל.

<u>Q6</u>

<u>סעיף א'- הגדרת הדקדוק:</u>

$\sum$ (Terminals)={ a,b,c,d,+,-,*,/,//,**,%,() }

V(variables)={ **<u>S</u>**,A,B,C,D,E,F,G,H,I,J,K,L,M } (S is the starting variable)

R(set rules):= (the symbol | refer to or)

S→AEBGFC|FBJFD|MIDECJD

A→a|B

B→b|C

C→c|D

D→d|A

E→+

F→-

G→*

H→/

I→//

J→**

K→%

L→()

M→(a+b)

G=(V, $\sum$, R, S)

סעיף ב'- שרשרת הגזירה:

.a שרשרת הגזירה תהיה כדלקמן:
S→AEBGFC→aEBGFC→a+BGFC→a+bGFC→a+b*FC→a+b*-C→**a+b*-c**

.b שרשרת הגזירה תהיה:
S→FBJFD→-BJFD→-bJFD→-b**FD→-b**-D→**-b**-d**

.c שרשרת הגזירה תהיה:
S→MIDECJD→(a+b)IDECJD→(a+b)//DECJD→(a+b)//dECJD→
(a+b)//d+CJD→(a+b)//d+cJD→(a+b)//d+c**D→**(a+b)//d+c**d**