<u>תרגיל בית מספר 6 - להגשה עד 18.6.21 בשעה 23:55</u>

קיראו בעיון את הנחיות העבודה וההגשה המופיעות באתר הקורס, תחת התיקייה assignments. חריגה מההנחיות תגרור ירידת ציון / פסילת התרגיל.

<u>הגשה</u>:

- תשובותיכם יוגשו בקובץ pdf ובקובץ py בהתאם להנחיות בכל שאלה.
- השתמשו בקובץ השלד skeleton6.py כבסיס לקובץ ה py אותו אתם מגישים.
  לא לשכוח לשנות את שם הקובץ למספר ת"ז שלכם לפני ההגשה, עם סיומת py.
- בסה"כ מגישים שני קבצים בלבד. עבור סטודנטית שמספר ת"ז שלה הוא 012345678 הקבצים שיש להגיש הם hw6_012345678.pdf ו- hw6_012345678.py.
- הקפידו לענות על כל מה שנשאלתם.
- תשובות מילוליות והסברים צריכים להיות תמציתיים, קולעים וברורים.
  להנחיה זו מטרה כפולה:
  1. על מנת שנוכל לבדוק את התרגילים שלכם בזמן סביר.
  2. כדי להרגיל אתכם להבעת טיעונים באופן מתומצת ויעיל, ללא פרטים חסרים מצד אחד אך ללא עודף בלתי הכרחי מצד שני. זוהי פרקטיקה חשובה במדעי המחשב.

שאלה 1 – Hashing

נתונה רשימה של n מחרוזות $[s_0, s_1, \dots, s_{n-1}]$, לאו דווקא שונות זו מזו. בנוסף נתון $k > 0$, וידוע שכל המחרוזות באורך לפחות k (ניתן להניח זאת בכל הפתרונות שלכם ואין צורך לבדוק או לטפל במקרים אחרים). אנו מעוניינים למצוא את כל הזוגות הסדורים של אינדקסים שונים (i, j), כך שקיימת חפיפה באורך k בדיוק בין רישא (התחלה) של s_i לסיפא (סיומת) של s_j. כלומר $s_i[:k] == s_j[-k:]$.

לדוגמה, אם האוסף מכיל את המחרוזות הבאות:

```
s0 = "a"*10
s1 = "b"*4 + "a"*6
s2 = "c"*5 + "b"*4 + "a"
```

אז עבור $k = 5$ יש חפיפה באורך k בין הרישא של s0 לבין הסיפא של s1, ויש חפיפה באורך k בין הרישא של s1 לבין הסיפא של s2. שימו לב שאנו לא מתעניינים בחפיפות אפשריות של מחרוזות עם עצמן, כמו למשל החפיפה באורך 5 בין רישא של s0 לסיפא של עצמה. לכן, הפלט במקרה זה יהיה שני הזוגות $(0,1)$ ו-$(1,2)$. אבל ייתכן שיש שתי מחרוזות זהות, ואז כן נתעניין בחפיפה כזו. למשל עבור s0=s1="aaa" ועבור $k = 1$ הפלט אמור להיות $(0,1)$ ו-$(1,0)$.

א. נציע תחילה את השיטה הבאה למציאת כל החפיפות הנ"ל: לכל מחרוזת נבדוק את הרישא באורך k שלה אל מול כל הסיפות באורך k של כל המחרוזות האחרות. ממשו את הפתרון הזה בקובץ השלד, בפונקציה **prefix_suffix_overlap(lst,k)**, אשר מקבלת רשימה (מסוג list של פייתון) של מחרוזות, וערך מספרי k, ומחזירה רשימה עם כל זוגות האינדקסים של מחרוזות שיש ביניהן חפיפה כנ"ל. אין חשיבות לסדר הזוגות ברשימה, אך יש כמובן חשיבות לסדר הפנימי של האינדקסים בכל זוג.

דוגמאות הרצה:

```
>>> s0 = "a"*10
>>> s1 = "b"*4 + "a"*6
>>> s2 = "c"*5 + "b"*4 + "a"
>>> prefix_suffix_overlap([s0,s1,s2], 5)
[(0, 1), (1, 2)] #could also be [(1, 2), (0, 1)]
```

ב. ציינו מהי סיבוכיות הזמן של הפתרון הזה במקרה הגרוע, כתלות ב-n וב-k במונחים של $O(\dots)$. הניחו כי השוואה בין שתי תת מחרוזות באורך k דורשת $O(k)$ פעולות במקרה הגרוע. ציינו גם מתי מתקבל המקרה הגרוע, בהנחה שהשוואת מחרוזות עוברת תו-תו בשתי המחרוזות במקביל משמאל לימין, ומפסיקה ברגע שהתגלו תווים שונים.

ג. כעת נייעל את המימוש ונשפר את סיבוכיות הזמן (בממוצע), ע"י שימוש במנגנון של טבלאות hash. לשם כך נשתמש במחלקה חדשה בשם Dict, שחלק מהמימוש שלה מופיע בקובץ השלד. מחלקה זו מזכירה מאוד את המחלקה Hashtable שראיתם בהרצאה, אבל ישנם שני הבדלים:

1) בקוד מההרצאה האיברים בטבלה הכילו רק מפתחות (keys), בדומה ל-set של פייתון, ואילו אנחנו צריכים לשמור גם מפתחות וגם ערכים נלווים (values), בדומה לטיפוס dict של פייתון. המפתחות במקרה שלנו יהיו רישות באורך k של המחרוזות הנתונות, ואילו הערך שנלווה לכל רישא כזו הוא האינדקס של המחרוזת ממנה הגיעה הרישא (מספר בין 0 ל-$n-1$). חישוב ה-hash לצורך הכנסה וחיפוש במילון מתבצע על המפתח בלבד.
2) מכיוון שיכולות להיות רישות זהות למחרוזות הנתונות, נרצה לאפשר חזרות של מפתחות ב-Dict (ראו בדוגמה בהמשך).

השלימו בקובץ השלד את המימוש של המתודה **`find(self, key)`** של המחלקה Dict, המתודה מחזירה רשימה (list של פייתון) עם **כל** ה-values שמתאימים למפתח key הנתון (לא חשוב באיזה סדר). אם אין כאלו תוחזר רשימה ריקה.
דוגמאות הרצה:

```
>>> d = Dict(3)
>>> d.insert("a", 56)
>>> d.insert("a", 34)
>>> d #calls __repr__
0 []
1 []
2 [['a', 56], ['a', 34]]

>>> d.find("a")
[56, 34] #order does not matter
>>> d.find("b")
[]
```

ד. השלימו את מימוש הפונקציה **`prefix_suffix_overlap_hash1(lst, k)`**, שהגדרתה זהה לזו של `prefix_suffix_overlap(lst, k)`, אלא שהיא תשתמש במחלקה Dict מהסעיף הקודם. כאמור, כל הרישות יוכנסו למילון תחילה, ואז נעבור על כל הסיפות ונבדוק לכל אחת אם היא נמצאת במילון.

ה. לצורך סעיף זה בלבד, הניחו כי אין שתי מחרוזות עם אותו סיפא, אותה רישא, או רישא של מחרוזת כלשהי ששווה לסיפא של מחרוזת כלשהי. בפרט, התנאי האחרון מבטיח שהפלט של `prefix_suffix_overlap` יהיה רשימה ריקה (אין התאמות). ציינו מהי סיבוכיות הזמן של הפתרון מסעיף ד' **בממוצע** (על פני הקלטים שמקיימים את התנאי של סעיף זה), כתלות ב-n וב-k במונחים של $O(\ldots)$. הניחו כי השוואה בין שתי תת מחרוזות באורך k דורשת $O(k)$ פעולות במקרה הגרוע, וכך גם חישוב hash על מחרוזת באורך k נמקו את תשובתכם בקצרה.

אוניברסיטת תל אביב - בית הספר למדעי המחשב
מבוא מורחב למדעי המחשב, אביב 2021

<u>שאלה 2 – גנרטורים</u>

בשאלה זו נשתמש בגנרטורים כדי לקרב את π בקירובים הולכים ומשתפרים, בדרכים שונות.

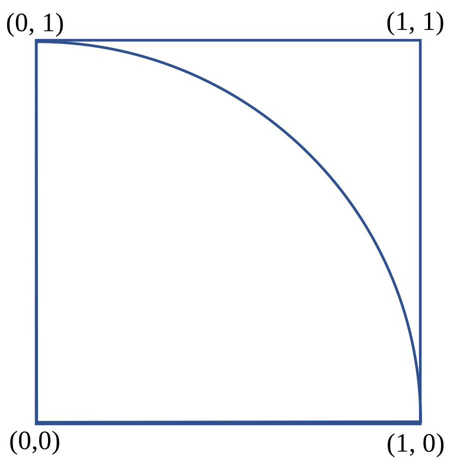
הנחיות:

- בפתרון השאלה מומלץ להיעזר בהרצאה 15 על חישוב נומרי.
- בשאלה זו אין להשתמש בספריית המתמטיקה של פייתון (math) בקוד שאתם מגישים.
- בכל הסעיפים **יש להשתמש בפונקציות מסעיפים קודמים** היכן שאפשר וזה מתבקש.
- שימו לב: שלושת הגנרטורים שמתחילים בשם _pi_approx צריכים להחזיר קירוב ל-π ולא ל $\frac{\pi}{4}$ או ל $\frac{\pi}{2}$.

א. השלימו בקובץ השלד את פונקציית הגנרטור ()powers_of_2, אשר מחזירה גנרטור שבכל קריאה אליו (next) מחזיר את החזקה הבאה של 2, החל מ-2^0. אין להשתמש בפונקציה באופרטור החזקה. דוגמת הרצה:

```
>>> gen = powers_of_2()
>>> print([next(gen) for i in range(10)])
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

ב. השלימו בקובץ השלד את פונקציית הגנרטור ()pi_approx_monte_carlo, אשר מחזירה גנרטור שמחזיר קירוב ל-π. בכל קריאה לגנרטור (next), הגנרטור יגריל נקודות במישור, באמצעות פונקציית ()random.random, כך שערכי x, y שלה שניהם בטווח [0, 1) (כלומר הגרלת נקודה בתוך ריבוע באופן יוניפורמי). הגנרטור יספור כמה מסך הנקודות נפלו בתוך רבע עיגול שמרכזו בראשית ורדיוסו באורך 1. יחס הנקודות בתוך העיגול חלקי כלל הנקודות נותן קירוב ליחס השטחים של רבע העיגול חלקי הריבוע שחוסם אותו. באמצעות חישוב זה הגנרטור יחזיר בכל קריאה אליו קירוב הולך ומשתפר (בתוחלת) ל-π.



$$\frac{N_{inside}}{N_{total}} \approx \frac{Area_{quarter-circle}}{Area_{square}} = \frac{\frac{\pi r^2}{4}}{r^2} = \frac{\pi}{4}$$

בכל קריאה לגנרטור הוא יגריל מספר הולך וגדל אקספוננציאלית של נקודות. בקריאה הראשונה הוא יגריל נקודה אחת ויחזיר קירוב שניתן בעזרת נקודה זו. בקריאה השנייה הוא יגריל **עוד** 2 נקודות ויחזיר קירוב שניתן בעזרת הגרלת 3 נקודות סה"כ. בקריאה השלישית השלישית יגריל **עוד** 4 וכו'.

ג. בסעיף זה נקרב את π באמצעות הנוסחה של לייבניץ:

$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots = \frac{\pi}{4}$$

a. השלימו בקובץ השלד את פונקציית הגנרטור ()leibniz, אשר מחזירה גנרטור שמחזיר את האיברים בטור האינסופי הנ"ל באופן סדרתי (כלומר בכל קריאה מחזיר איבר בסכום). דוגמת הרצה:

```
>>> gen = leibniz()
>>> [next(gen) for i in range(4)]
[1.0, -0.3333333333333333, 0.2, -0.14285714285714285]
```

b. השלימו בקובץ השלד את פונקציית הגנרטור infinite_series(gen), שבהינתן גנרטור gen, מחזירה גנרטור שסוכם את איברי gen ומחזיר את הסכום אחרי כל קריאה.

c. השלימו בקובץ השלד את פונקציית הגנרטור ()pi_approx_leibniz, אשר מחזירה גנרטור שמחזיר קירוב הולך ומשתפר ל-π בכל קריאה לו. גם גנרטור זה מוסיף מספר הולך וגדל אקספוננציאלי של איברים. כלומר, בקריאה הראשונה הוא יחזיר קירוב של π בעזרת איבר אחד מנוסחת לייבניץ, בקריאה השני הוא **יוסיף לסכום עוד** 2 איברים, בקריאה השלישית **עוד** 4 איברים וכו'.

ד. בסעיף זה נקרב את π באמצעות חישוב אינטגרל.

a. השלימו בקובץ השלד את פונקציית הגנרטור ()unit_slicing, שמחזירה גנרטור שבכל קריאה מחזיר רשימת מספרים המתארת חלוקה יוניפורמית של הקטע $[0,1)$. בכל קריאה תחזור חלוקה שמחלקת את הקטע לפי 2 יותר איברים מהחלוקה הקודמת. כל חלק בחלוקה מוגדר ע"י הנקודה השמאלית שלו. דוגמת הרצה:

```
>>> gen = unit_slicing()
>>> for i in range(4):
...     print(next(gen))
[0.0]
[0.0, 0.5]
[0.0, 0.25, 0.5, 0.75]
[0.0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875]
```

b. השלימו בקובץ השלד את פונקציית הגנרטור integral(func, a, b), שמחזירה גנרטור שבכל קריאה מחזיר קירוב לאינטגרל $\int_a^b func(x)dx$. הקירוב ייעשה באמצעות סכימת שטחים של מלבנים, כפי שראינו בהרצאה על חישוב נומרי. בקריאה הראשונה הוא יחזיר קירוב לאינטגרל בעזרת מלבן אחד, $(b-a)\cdot func(a)$, ובכל קריאה נוספת הוא יחזיר קירוב לאינטגרל בעזרת פי 2 יותר מלבנים מאשר בקריאה הקודמת **בסך הכל**. (שימו לב- בניגוד לקירובים

הקודמים, גנרטור זה לא משתמש בחישובים שעשה בקריאות הקודמות לו, אלא **מחשב מחדש בכל קריאה**.)

c. השלימו בקובץ השלד את פונקציית הגנרטור ()pi_approx_integral, אשר מחזירה קירוב הולך ומשתפר ל-π באמצעות האינטגרל הבא:

$$\int_{-1}^{1} \sqrt{1-x^2}\,\mathrm{d}x = \arcsin 1 = \frac{\pi}{2}$$

ה. הפעילו את כל אחד משלושת הגנרטורים שמימשתם בסעיפים הקודמים 20 פעמים, דווחו מה הקירוב שקיבלתם ומה השגיאה המוחלטת שלו מהערך של math.pi. דרגו את שלוש השיטות מהטובה לגרועה ביותר.

<u>שאלה 3 – קוד האפמן</u>

א. מצאו את קוד האפמן האופטימלי עבור הקורפוס (corpus) הבא:
a: 1 b: 1 c: 2 d: 3 e: 5 f: 8 g: 13 h: 21

סדרת התדירויות הנ"ל מבוססת על 8 מספרי פיבונאצ'י הראשונים.

ב. הכלילו את תשובתכם מסעיף א' למציאת קוד האפמן אופטימלי כאשר התדירויות הן n מספרי פיבונאצ'י הראשונים. נמקו בקצרה, ללא צורך בהוכחה מפורטת, מדוע הכללה זו נכונה.

ג. נתון קובץ שמכיל תווים מתוך אלפבית בן 256 תווים. בנוסף נתון קורפוס עם תדירויות: $a_1 < a_2 < \cdots < a_n$ (כאשר $n = 256$) ומתקיים: $a_n < 2a_1$.

תהי a_1 תדירות התו p (התדירות המינימלית), ותהי a_n תדירות התו q (התדירות המקסימלית).
יהיו $C(a_1)$, $C(a_n)$ קודי ההאפמן שמתקבלים עבור התווים q, p בהתאמה.

מהו ההפרש בין $|C(a_1)|$ (מספר הביטים שדרושים כדי לקודד את התו p) לבין $|C(a_n)|$ (מספר הביטים שדרושים כדי לקודד את התו q)?

על תשובתכם להיות מנומקת!

ד. כיצד תשתנה תשובתכם לסעיף ג אם עכשיו n = 300? הסבירו בקצרה.

ה. כיצד תשתנה תשובתכם לסעיף ג אם נתון ש n = 272 ובנוסף שהתדירויות מקיימות את התנאים הבאים:

a. $a_{16} < 2a_1$

b. $a_{272} < 2a_{17}$

c. $16a_{16} < a_{17}$

הסבירו את תשובתכם ושרטטו סקיצה שממחישה את מבנה עץ האפמן עבור קורפוס זה.

המלצה: ברוב הסעיפים ניתן להריץ דוגמאות קוד כדי להשתכנע שהתשובה שלכם נכונה. אל תפספסו את ההזדמנות לוודא זאת.

<u>שאלה 4 – למפל זיו</u>

השאלה עוסקת בשינוי באלגוריתם למפל-זיו לדחיסת טקסט.

כזכור, הפונקציה LZW_compress מחזירה את ייצוג הביניים של דחיסת למפל-זיו של המחרוזת text. למשל:

```
>>> LZW_compress("abcdabc")
['a', 'b', 'c', 'd', [4, 3]]
>>> LZW_compress("abab")
['a', 'b', 'a', 'b']
>>> LZW_compress("ababab")
['a', 'b', [2, 4]]
```

בנוסף, ראינו בכיתה את הפונקציה:

```
def inter_to_bin(intermediate, W=2**12-1, L=2**5-1)
```

שבהינתן רשימה lst עם ייצוג ביניים של מחרוזת דחוסה, מחזירה מחרוזת של ביטים, המייצגת את רצף הביטים לאחר הדחיסה. נזכיר, שתו שלא נדחס ייוצג ע״י הביט 0 ואחריו 7 ביטים עבור התו עצמו (סה״כ 8 ביטים, כלומר גם בתרגיל זה אנחנו מניחים לשם פשטות כי אנחנו מטפלים רק בתווי ASCII), ואילו מקטע שנדחס ייוצג ע״י הביט 1 ואחריו 12 ביטים עבור ההיסט אחורה, ו- 5 ביטים עבור אורך המקטע שנדחס (סה״כ 18 ביטים). שימו לב שבחישוב זה לקחנו בחשבון את ערכי ברירת המחדל של הפרמטרים W,L של שתי הפונקציות.

דוגמאות הרצה:

```
>>> inter_to_bin(LZW_compress("abcdabc"))
'01100001011000100110001101100100100000000010000011'
>>> len(inter_to_bin(LZW_compress("abcdabc")))
50      # 4*8 + 18
>>> inter_to_bin(LZW_compress("abab"))
'01100001011000100110000101100010'
>>> len(inter_to_bin(LZW_compress("abab")))
32      # 4*8
```

בעמוד הבא מופיעה הפונקציה **LZW_compress_new** שמציגה מימוש של אלגוריתם שונה במעט עבור דחיסת למפל זיו.
ענו בקובץ ה pdf על הסעיפים הבאים ביחס ל: **LZW_compress_new, LZW_compress.**

א. תנו דוגמא למחרוזת s המקיימת:

LZW_compress(s) = LZW_compress_new(s).

מה יהיה הפלט (ייצוג הביניים) שיתקבל בשתי ההרצות?

ב. טענה: קיימת מחרוזת **s** שמקיימת:

```
len(inter_to_bin(LZW_compress_new(s))) <
len(inter_to_bin(LZW_compress(s)))
```

תנו דוגמא למחרוזת **s** כזו בצירוף שני ייצוגי הביניים המתקבלים ע״י הפעלת כל אחת מהפונקציות הנ״ל, או הסבירו מדוע אין מחרוזת **s** כזו.

ג. טענה: קיימת מחרוזת **s** שמקיימת:

```
len(inter_to_bin(LZW_compress_new(s))) >
len(inter_to_bin(LZW_compress(s)))
```

תנו דוגמא למחרוזת **s** כזו בצירוף שני ייצוגי הביניים המתקבלים ע״י הפעלת כל אחת מהפונקציות הנ״ל, או הסבירו מדוע אין מחרוזת **s** כזו.

```
def LZW_compress_new(text, start=0, W=2**12-1, L=2**5-1):
    n = len(text)
    if start >= n:
        return []

    #find the maximal length matching
    m,k = maxmatch(text, start, W, L)

    res1 = [text[start]] + \
          LZW_compress_new(text, start+1, W, L)
    res1_len = len(inter_to_bin(res1, W, L))

    if k < 3:
        return res1

    res2 = [[m,k]] + LZW_compress_new(text, start+k, W, L)
    res2_len = len(inter_to_bin(res2, W, L))

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

<u>שאלה 5 – קודים לתיקון שגיאות</u>

יהא $C: \{0,1\}^k \to \{0,1\}^n$ קוד לתיקון שגיאות. נאמר כי גנרטור g מייצר את C אם אוסף האיברים שהגנרטור מייצר הוא תמונת הקוד וכל איבר נוצר בדיוק פעם אחת (שימו לב: אין חשיבות לסדר יצור האיברים).

א. עליכם לממש את הפונקציה dist(code_gen, n), המקבלת כקלט פונקציית גנרטור code_gen ומספר שלם חיובי n, ומחזירה את המרחק של הקוד C.
מובטח כי פונקציית הגנרטור code_gen מחזירה גנרטור המייצר קוד $C: \{0,1\}^k \to \{0,1\}^n$. כלומר, הפקודה g = code_gen() מחזירה גנרטור g המייצר קוד $C: \{0,1\}^k \to \{0,1\}^n$.

הנחיות:
1. על הפונקציה לפעול בזיכרון $O(n)$
2. ניתן להניח כי גנרטור הנוצר מהפעלה של הפונקציה code_gen דורש $O(n)$ זיכרון
3. שימו לב: code_gen היא **<u>פונקצית גנרטור</u>** ולא גנרטור.
4. יש לכתוב את המימוש בקובץ ה-PDF שאתם מגישים, ולא בקובץ השלד.

ב. בסעיף זה ננתח את זמן הריצה של הפונקציה שכתבתם בסעיף א' כפונקציה של n, k.
ניתן להניח כי שליפה של איבר בודד מהגנרטור הנוצר מהפעלה של הפונקציה code_gen מתבצעת בזמן $O(n)$

איזה מהמשפטים הבאים מתאר בצורה הנכונה וההדוקה ביותר את סיבוכיות זמן הריצה של הפונקציה dist?
1. זמן הריצה אקספוננציאלי ב-n ואקספוננציאלי ב- k
2. זמן הריצה פולינומי ב-n ופולינומי ב-k
3. זמן הריצה אקספוננציאלי ב-k ופולינומי ב-n
4. זמן הריצה פולינומי ב-k ואקספוננציאלי ב-n

מצאו חסם הדוק על סיבוכיות הזמן של הפונקציה **והסבירו** את תשובתכם.

לכל אחת מהטענות הבאות יש לסמן האם היא נכונה או לא. אם הטענה נכונה, יש להסביר בקצרה מדוע. אם היא לא נכונה, יש לספק דוגמא נגדית.
תזכורות:
- אם A, B הן קבוצות, $A \cap B$ הוא החיתוך שלהן ו-$|A|$ הוא מספר האיברים ב-A.
- קוד (n, k, d) נקרא **לא טריוויאלי** אם $d > 1$.
- **כדור** ברדיוס r סביב $y \in \{0,1\}^n$ מוגדר כך: $B(y, r) = \{z \in \{0,1\}^n | \Delta(y, z) \le r\}$

ג. אם C הוא קוד (n, k, d) לא טריוויאלי אזי לכל $z \in \{0,1\}^n$ מתקיים $\left|B\left(z, \left\lfloor\frac{d-1}{2}\right\rfloor\right) \cap \mathrm{Im}C\right| = 1$.

ד. אם C הוא קוד (n, k, d) לא טריוויאלי ו-ℓ מספר שלם המקיים כי $\ell > \left\lfloor\frac{d-1}{2}\right\rfloor$, אזי לכל $z \in \{0,1\}^n$ מתקיים $|B(z, \ell) \cap \mathrm{Im}C| > 1$.

אוניברסיטת תל אביב - בית הספר למדעי המחשב
מבוא מורחב למדעי המחשב, אביב 2021

שאלה 6 – דקדוקים חסרי הקשר ו-CYK

א. **הגדירו דקדוק** חסר הקשר המתאר ביטויים (expressions) חשבוניים בשפת פייתון. על הדקדוק לתמוך בביטויים המכילים את המשתנים a, b, c, d, את פעולות החשבון הבאות: +, -, *, /, //, **, % וסוגריים עגולים.

הנחיות:

- להזכירכם, דקדוק מוגדר ע״י הרביעייה $G = (V, \Sigma, R, S)$. עליכם להגדיר **במפורש** את ארבעת האיברים ב-G.
- הדקדוק שלכם לא חייב להיות בפורמט CNF, אבל הוא כן חייב להיות חסר הקשר.
- לצורך פשטות, נניח כי אין רווחים בין התווים במחרוזות.

ב. לכל אחד מהביטויים הרשומים בהמשך, **הראו שרשרת גזירה** שלו בדקדוק שהגדרתם בסעיף א׳.
לדוגמה, עבור הדקדוק $G = (V = \{S, A, B\}, \Sigma = \{a, b\}, R = \{S \rightarrow AB, A \rightarrow a, B \rightarrow b\}, S)$
והמחרוזת "ab", שרשרת גזירה תיראה כך:

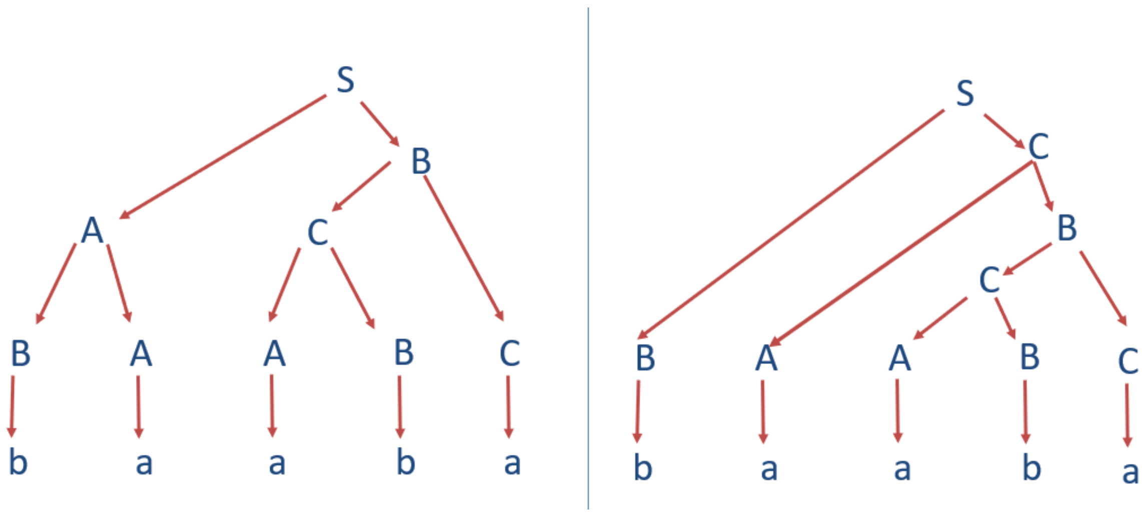
$$S \rightarrow AB \rightarrow aB \rightarrow ab$$

a. a+b*-c
b. -b**-d
c. (a+b)//d+c**d

ג. להזכירכם, בהינתן דקדוק G בצורת CNF ומחרוזת st, אלגוריתם CYK מחזיר האם המחרוזת יכולה להיגזר ע״י הדקדוק, או במילים אחרות, האם st בשפה של G: $st \in \mathcal{L}(G)$.
ראינו כי יש זוגות של (דקדוק, מחרוזת) עבורם יש יותר מעץ גזירה אחד שגוזר את המחרוזת. מצב זה נקרא ambiguity או עמימות.
בסעיף זה, נרצה לערוך שינוי קטן ב-CYK כך שבהינתן דקדוק ומחרוזת, האלגוריתם **יחזיר את העומק המינימלי של עץ גזירה** שגוזר את המחרוזת st, **או 1-** אם המחרוזת לא יכולה להיגזר ע״י הדקדוק.
האלגוריתם לאחר השינוי צריך להיות באותה סיבוכיות הזמן והמקום של אלגוריתם CYK המקורי.
לדוגמה, עבור הדקדוק

$$S \rightarrow AB \mid BC$$
$$A \rightarrow BA \mid a$$
$$B \rightarrow CC \mid b$$
$$C \rightarrow AB \mid a$$

והמחרוזת $st = "baaba"$, שני עצי הגזירה הנ״ל הם חוקיים:



אך עומק עץ הגזירה השמאלי הוא 4 ועומק עץ הגזירה הימני הוא 5. במקרה זה, 4 הוא העומק המינימלי של עץ גזירה שגוזר את המחרוזת, ולכן עבור דקדוק ומחרוזת אלו האלגוריתם שלכם צריך **להחזיר 4**.

עבור אותו הדקדוק ומחרוזת $st = "baab"$, האלגוריתם צריך **להחזיר 1-** מכיוון שהמחרוזת אינה בשפה של הדקדוק.

השלימו בקובץ השלד את הפונקציות CYK_d, fill_cell_d, fill_length_1_cells_d בהתאם. מומלץ להשתמש בקוד המקורי של CYK ולשנותו לפי הצורך.

דוגמאות הרצה:

```
>>> rule_dict = {"S": {"AB", "BC"}, "A": {"BA", "a"}, "B": {"CC",
"b"}, "C": {"AB", "a"}}
>>>> CYK_d("baaba", rule_dict, "S")
4
>>>> CYK_d("baab", rule_dict, "S")
-1
```

<u>סוף</u>