אוניברסיטת תל אביב - בית הספר למדעי המחשב
מבוא מורחב למדעי המחשב, אביב 2024

<u>תרגיל בית מספר 5 - להגשה עד 13/08/2024 בשעה 23:59</u>

קראו בעיון את הנחיות העבודה וההגשה המופיעות באתר הקורס. חריגה מההנחיות תגרור ירידת ציון / פסילת התרגיל.

<u>הנחיות לצורת ההגשה</u>:

- תשובותיכם יוגשו בקובץ pdf ובקובץ py בהתאם להנחיות בכל שאלה.
- בסה״כ מגישים שני קבצים בלבד. עבור סטודנטית שמספר ת״ז שלה הוא 012345678 הקבצים שיש להגיש הם hw5_012345678.pdf ו- hw5_012345678.py.
- השתמשו בקובץ השלד skeleton5_2024a.py כבסיס לקובץ אותו אתם מגישים.
- לא לשכוח לשנות את שם הקובץ למספר ת״ז שלכם לפני ההגשה, עם סיומת py.

<u>הנחיות לפתרון</u>:

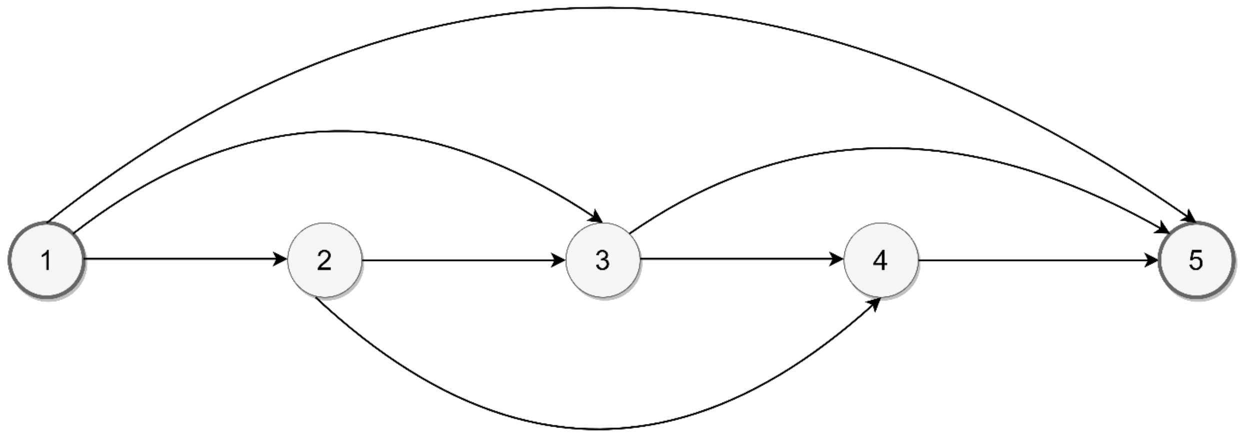
- הקפידו לענות על כל מה שנשאלתם.
- בכל שאלה, אלא אם מצוין אחרת באופן מפורש, ניתן להניח כי הקלט תקין.
- אין להשתמש בספריות חיצוניות פרט לספריות time, math, random אלא אם נאמר במפורש אחרת.
- תשובות מילוליות והסברים צריכים להיות תמציתיים, קולעים וברורים.
  להנחיה זו מטרה כפולה:
  1. על מנת שנוכל לבדוק את 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.
- כיוון שלמדנו בשבועות האחרונים כיצד לנתח את זמן הריצה של הקוד שלנו, החל מתרגיל זה ולאורך שארית הסמסטר (וכן במבחן) נדרוש שכל הפונקציות שאנו מממשים תהיינה יעילות ככל הניתן. לדוגמה, אם ניתן לממש פתרון לבעיה בסיבוכיות $O(logn)$, ואתם מימשתם פתרון בסיבוכיות $\Theta(n)$, תקבלו ניקוד חלקי על הפתרון.
- בשאלות שבהן ישנה דרישה לניתוח סיבוכיות זמן הריצה, הכוונה היא לסיבוכיות זמן הריצה של המקרה הגרוע ביותר (worst-case complexity). כמו כן, אלא אם כן צוין אחרת, ניתן להגיש פתרונות שרצים בזמן <u>יעיל יותר</u> מהדרישה בתרגיל. (לדוגמא, אם נדרש שסיבוכיות הזמן של הפתרון תהיה $O(n^2)$, ניתן להגיש קוד שסיבוכיות זמן הריצה שלו היא $O(n)$).
- את שאלות 2 ו 5 יהיה ניתן לפתור בשבוע האחרון, לאחר התרגול המתאים.

אוניברסיטת תל אביב - בית הספר למדעי המחשב
מבוא מורחב למדעי המחשב, אביב 2024

שאלה 1

<u>הדרכה</u>: באתר המודל של הקורס, תחת "פתרונות והדרכות" מצורפים 4 סרטוני הדרכה שיעזרו לכם בהבנת התרגיל. מומלץ ביותר לצפות בהם לפני שאתם מתחילים לפתור את התרגיל.

נגדיר מבנה נתונים חדש: **<u>רשימה מקושרת לוגריתמית</u>**. המבנה החדש מתבסס על הרשימה המקושרת שראינו בהרצאה ובתרגול עם שינוי מרכזי – במקום שכל צומת יחזיק מצביע לצומת הבא אחריו, כל צומת מחזיק מצביעים לצמתים שנמצאים 2^i צעדים אחריו, לכל $0 \le i \le \log n$ (אם ישנם כאלו), כאשר n הוא מספר האיברים ברשימה.
להלן דיאגרמה לרשימה מקושרת לוגריתמית בת 5 איברים (כרגיל, איברי הרשימה מופיעים משמאל לימין). שימו לב למשל שלצומת מספר 1 יש 3 מצביעים קדימה (לצמתים 2, 3 ו-5), לצומת 3 יש 2 מצביעים קדימה בלבד (לצמתים 4 ו-5) ואילו לצומת 5 אין מצביעים קדימה בכלל (כיוון שהוא האחרון ברשימה).
הבהרה: אין חשיבות למיקום החצים בתרשים ביחס לצמתים (כלומר, האם הם מעל, לצד או מתחת לצמתים ברשימה) ולמספר המופיע בתוך הצומת. אלמנטים אלו הם לנוחות הקריאה של הדוגמא בלבד.



כדי לממש את המבנה החדש בפייתון, נייצג את רשימת המצביעים של כל צומת על ידי שדה בשם next_list מטיפוס רשימה של פייתון (list). האיבר באינדקס i ברשימת המצביעים next_list יהיה המצביע לצומת שנמצא 2^i צמתים אחרי הצומת הנוכחי.
בצומת האחרון ברשימה המקושרת שדה ה-next_list יהיה רשימה ריקה (זאת בשונה מרשימה מקושרת כפי שראינו בכיתה, בה שדה ה-next של הצומת האחרון הינו None).
להלן המחלקה של צומת ברשימה מקושרת לוגריתמית:

```
class LLLNode:
    def __init__(self, val):
        self.next_list = []
        self.val = val
```

רשימה מקושרת לוגריתמית תיוצג כרגיל על ידי שדה head שיצביע לראש הרשימה ושדה len בו נשמור את אורך הרשימה. להלן המחלקה של רשימה מקושרת לוגריתמית עם מתודת אתחול וחישוב אורך:

```
class LogarithmicLinkedList:
    def __init__(self):
        self.head = None
        self.size = 0

    def __len__(self):
        return self.size
```

כדוגמא קונקרטית, אם נסמן ב-p1, p2, p3, p4, p5 את האיברים מטיפוס LLLNode שמייצגים את צמתי הרשימה שבציור (משמאל לימין) ונקרא לרשימה בה הם נמצאים L אזי למשל:
`p1.next_list = [p2, p3, p5]`, `p5.next_list = []` וכן `L.head = p1`.

<u>הנחיה</u>: את סעיפים א׳ וב׳ יש לממש בסיבוכיות זמן $O(\log n)$ כאשר n הוא מספר האיברים ברשימה.

<u>סעיף א׳</u>
ממשו את המתודה add של המחלקה LogarithmicLinkedList. המתודה תקבל כקלט רשימה מקושרת לוגריתמית self ומשתנה נוסף val. המתודה תוסיף צומת חדש **<u>לתחילת הרשימה</u>** self שערכו הוא val כך שיעמוד בהגדרת המחלקה מהעמוד הקודם. שימו לב כי בקובץ השלד כבר נתון חלק מהמימוש.

<u>סעיף ב׳</u>
ממשו את המתודה __getitem__. המתודה תקבל כקלט רשימה מקושרת לוגריתמית self באורך n ומשתנה נוסף $0 \leq i < n$ ותחזיר את הערך בצומת ה-i ברשימה self.

<u>סעיף ג׳ (רשות)</u>
בסעיף זה נניח כי ערכי הרשימה (כלומר, ערכי ה-val של כל צומת ברשימה) ממוינים בסדר עולה. לפניכם מימוש למתודה __contains__. המתודה מקבלת כקלט רשימה מקושרת לוגריתמית self ומשתנה val ומחזירה True אם״ם יש צומת ברשימה שערכו הוא val.

```
def __contains__(self, val):
    p = self.head
    k = 1
    while k != 0:
        if p.val == val:
            return True
        k = 0
        m = len(p.next_list)
        while k < m and p.next_list[k].val <= val:
            k += 1
        if k > 0:
            p = p.next_list[k-1]
    return False
```

בסרטון שהועלה למודל בו אנחנו מסבירים על רשימות לוגריתמיות, ראינו שזמן הריצה של מתודה זו הוא $\Theta(\log^2 n)$, ודיברנו על שיפור פשוט שיביא לזמן ריצה של $O(\log n \cdot \log\log n)$. שפרו את המימוש הנתון למתודה __contains__ כך שזמן הריצה שלה יהיה $O(\log n)$.

אוניברסיטת תל אביב - בית הספר למדעי המחשב
מבוא מורחב למדעי המחשב, אביב 2024

<u>שאלה 2 – גנרטורים</u>

<u>הגדרה</u>: גנרטור הוא **<u>בעל השהייה סופית</u>** (finite delay) אם כל קריאה ל-next עליו מסתיימת תוך זמן סופי (לא משנה כמה זמן). כל קריאת next תמיד מחזירה איבר או שגיאת StopIteration בפרק זמן סופי. שימו לב שגם גנרטור שמייצר סדרה אינסופית יכול להיות בעל השהייה סופית.

<u>הגדרה</u>: גנרטור g **<u>מייצר את הקבוצה</u>** S (סופית או אינסופית) אם:

1. g הוא בעל השהייה סופית.
2. $x \in S$ אמ"ם g מייצר את x לאחר מספר סופי של קריאות next.
3. g לא מייצר חזרות (כלומר, כל איבר ש-g מייצר הוא ייחודי).

בהגדרה זו אין חשיבות לסדר החזרת איברי S.

<u>הגדרה</u>: גנרטור g **<u>מייצר את הסדרה</u>** $\{a_n\}$ (סופית או אינסופית) אם g בעל השהייה סופית, וכן לאחר i קריאות next יוחזר הערך a_i (או StopIteration לאחר החזרת כל איברי הסדרה). שימו לב שכאן יש חשיבות לסדר החזרת איברי הסדרה.

לכל אחד מהגנרטורים הבאים, אם ניתן לבנות אותו כך שתהיה לו השהייה סופית, השלימו את פונקציית הגנרטור המתאימה בקובץ השלד. אחרת, הסבירו בקצרה מדוע לא ניתן לבנות גנרטור כזה בעל השהייה סופית.

א. ()gen1 המייצר את הקבוצה $\mathbb{Z}^2$, כלומר כל זוגות המספרים **<u>השלמים</u>**.
<u>תזכורת</u>: ראיתם בתרגול שאלה דומה עבור הקבוצה $\mathbb{N}^2$. ניתן להיעזר בקוד שראיתם.

ב. gen2(g) המקבל גנרטור g שמייצר סדרת מספרים כלשהי $\{a_n\}$ (g סופי או אינסופי, בעל השהייה סופית), מייצר את סדרת הסכומים החלקיים של איברי g. כלומר אם g מייצר את הסדרה $a_1, a_2, a_3, \ldots$ אז הגנרטור מייצר את הסדרה $a_1, a_1+a_2, a_1+a_2+a_3, \ldots$.

ג. gen3(g) המקבל גנרטור g שמייצר סדרת מספרים כלשהי $\{a_n\}$ (g סופי או אינסופי, בעל השהייה סופית), ומייצר את קבוצת איברי g שגדולים מ-0.

ד. gen4(g) המקבל גנרטור g שמייצר סדרת מספרים כלשהי $\{a_n\}$ (g סופי או אינסופי, בעל השהייה סופית), ובקריאת next ה-i מחזיר True אמ"ם איברי הסדרה $a_1, \ldots, a_i$ מהווים סדרה מונוטונית עולה או יורדת במובן החלש. כלומר, אמ"ם אחד משני הבאים מתקיים:

1) $a_1 \leq \cdots \leq a_i$
2) $a_1 \geq \cdots \geq a_i$

ה. gen5(g1, g2) המקבל שני גנרטורים (סופיים או אינסופיים, בעלי השהייה סופית) ומייצר את החיתוך שלהם, כלומר את כל קבוצת האיברים שמיוצרים גם על ידי g1 וגם על ידי g2.

ו. gen6(g1, g2) המקבל שני גנרטורים שמייצרים את הסדרות $\{a_n\}$ ו-$\{b_n\}$ (g1,g2 או שניהם סופיים ובאותו הגודל, או שניהם אינסופיים, בעלי השהייה סופית), ומייצר את סדרת האיברים המקיימים $a_i \neq b_i$.

ז. **()gen7 (רשות)**: גנרטור המייצר את סדרת הגנרטורים כך שהגנרטור ה i מחזיר את קבוצת כל המספרים שמתחלקים ב i. פורמלית נגדיר כי הגנרטור מייצר את הסדרה $\{G_i\}_{i=1,2,\ldots}$ כך ש G_i הוא גנרטור המייצר את הקבוצה הבאה $\{x_j \in \mathbb{N} \mid x_j mod\ i = 0\}$

אוניברסיטת תל אביב - בית הספר למדעי המחשב
מבוא מורחב למדעי המחשב, אביב 2024

<u>שאלה 3</u>

הערה: בשאלה זו נעסוק בעצי חיפוש בינאריים. לאורך השאלה נתעלם משדה ה-val של צמתים בעץ.
בניתוח סיבוכיות נחשיב פעולות על מספרים כלוקחות זמן קבוע.

שימו לב שבמחלקה BinarySearchTree מומשה המתודה __repr__ לנוחיותכם כך שתדפיס את העץ בצורה ברורה.

<u>סעיף א'</u>

בהינתן n1, n2 (n1<=n2), שני מפתחות של צמתים בעץ חיפוש בינארי, נגדיר את **<u>הצומת המחבר הראשון</u>** שלהם להיות הצומת העמוק ביותר כך שגם n1 וגם n2 הם חלק מתת העץ שתחתיו. למשל, עבור העץ t שבדוגמת ההרצה של סעיף ג', הצומת המחבר הראשון של 1 ו-3 הוא 2, והצומת המחבר הראשון של 4 ו-6 הוא 4.

ממשו את הפונקציה lowest_common_ancestor(t, n1, n2) המקבלת עץ מהמחלקה BinarySearchTree ושני מפתחות של צמתים בעץ החיפוש הבינארי ומחזירה את הצומת המחבר הראשון שלהם.
<u>הנחיות</u>: על המימוש להיות רקורסיבי. ניתן להשתמש בשדות הפנימיים של TreeNode ו-BinarySearchTree.

נתחו בקובץ ה-pdf את זמן הריצה של המתודה כפונקציה של n, גודל העץ.

<u>הגדרה</u>: עץ בינארי הוא **<u>מושלם</u>** אם לכל צומת שאינו עלה יש בדיוק 2 ילדים וכל עלי העץ בעומק זהה.

<u>סעיף ב' (רשות)</u>

שימו לב: בסעיף זה אנו מתעלמים מתוכן העץ – מפתח ו/או שדה ומתייחסים למבנה העץ בלבד.

הוכיחו את שתי הטענות הבאות בקובץ ה-pdf:

- הוכיחו כי לכל $n \geq 1$ קיים עץ בינארי **<u>מושלם</u>** בן n צמתים אם ורק אם קיים שלם $d \geq 1$ כך ש-$n = 2^d - 1$.
- הוכיחו כי במקרה זה העץ המושלם הוא יחיד.

<u>סעיף ג'</u>

ממשו את הפונקציה build_balanced המקבלת כקלט שלם חיובי $d \geq 1$. הפונקציה תחזיר כפלט **<u>עץ חיפוש בינארי מושלם</u>** בעומק d (כלומר, אובייקט מהמחלקה BinarySearchTree), שמפתחותיו הם המספרים: $1,2,3,\dots,2^d - 1$.

<u>הנחיות</u>: על המימוש להיות רקורסיבי. יש להשתמש בשדות הפנימיים של TreeNode ו-BinarySearchTree.

דוגמת הרצה:

```
>>> t = build_balanced(3)
>>> print(t)
                 4
        _______/ \_______
       2                 6
   __/ \__           __/ \__
  1       3         5       7
 / \     / \       / \     / \
#   #   #   #     #   #   #   #

>>> t.size
7
```

נתחו בקובץ ה-pdf את זמן הריצה של הפונקציה build_balanced כפונקציה של n, גודל העץ. הסבירו את תשובתכם.

<u>סעיף ד'</u>

ממשו את הפונקציה subtree_sum המקבלת כקלט עץ חיפוש בינארי מושלם שמפתחותיו הם המספרים $1,2,3,\ldots,2^d-1$ (עבור עומק עץ d) ומפתח בעץ. הפונקציה תחזיר כפלט את סכום המפתחות בתת העץ שתחת הצומת עם המפתח שקיבלנו בקלט (כולל אותו צומת).

למשל, עבור הדוגמה מסעיף ג':

```
>>> print(subtree_sum(t, 6))
>>> 18
```

נתחו בקובץ ה-pdf את זמן הריצה של הפונקציה subtree_sum כפונקציה של n, גודל העץ. הסבירו את תשובתכם.

<u>שאלה 4</u>

נתונה רשימה של n מחרוזות $[s_0, s_1, \ldots, s_{n-1}]$, 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-$(1,0)$.

סעיף 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)]
```

סעיף 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.

סעיף 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")
[]
```

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.

סעיף ד'

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 $O(\dots)$. 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.

שאלה 5 – דחיסה

סעיף א'

נתון קורפוס corpus באורך n מעל א"ב בן $t \geq 2$ תווים $\{a_1, \dots, a_t\}$ כאשר כל תו מופיע בקורפוס לפחות פעם אחת.

בנוסף, נתון עץ האפמן H שנוצר כתוצאה מהרצת האלגוריתם של האפמן לבניית עץ על הקורפוס corpus.

עבור כל אחד מארבעת הפריטים שלפניכם, ציינו בקובץ ה-PDF את ערכו כפונקציה של t, n, והסבירו בקצרה את תשובתכם. אם ישנו ערך יחיד, ציינו אותו במפורש. אם ישנו טווח ערכים אפשרי, תנו ערך תחתון וערך עליון **הדוקים ככל הניתן**. שימו לב שיש לתת תשובות מדויקות, ולא במונחי $O(\cdot)$ או $\Theta(\cdot)$.

תזכורת: גובה של עץ הוא אורך מסלול ארוך ביותר **בקשתות** מהשורש לעלה כלשהו בעץ. כמו כן משקל של עלה בעץ הוא שכיחות התו המתאים בקורפוס (היזכרו כיצד הוגדר בכיתה משקל של צומת פנימי בעץ).

i. מספר העלים בעץ H
ii. משקל השורש של העץ H
iii. גובה העץ H

.iv מספר הצמתים בעץ H

<u>סעיף ב'</u>

נתון קורפוס עם תדירויות $a_1 < a_2 < \cdots < a_n$ כאשר $n = 256$ (כלומר יש 256 תווים שונים בקורפוס). כמו כן, מתקיים $a_n < 2a_1$. לאחר הרצת האלגוריתם לייצור עץ האפמן מהקורפוס, נסמן ב-ℓ את כמות הביטים בקידוד של התו שתדירותו a_1, וב-u את כמות הביטים בקידוד של התו שתדירותו a_n. מהו ההפרש בין ℓ ל-u?

<u>סעיף ג'</u>

באלגוריתם למפל-זיו שראינו בכיתה, כל תו בודד קודד על ידי הביט 0 ואחריו 7 ביטים עבור ייצוג ה-ASCII שלו. בסעיף זה, נרצה לשפר ולקודד תווים בודדים באמצעות קוד האפמן, במקום קוד ASCII. את החזרות נמשיך לקודד באופן המקורי, וכן נמשיך להשתמש בביט אינדיקטור כדי להבדיל בין בלוק המייצג תו לבין בלוק המייצג חזרה.

<u>תזכורת</u>: נאמר שקוד c הוא קוד האפמן אם קיים טקסט corpus כלשהו כך שהרצת האלגוריתם של האפמן על corpus תניב את הקוד c. הקוד c מיוצג ע"י מילון (של פייתון) כפי שראינו בכיתה.

.i ממשו את הפונקציה LZW_compress_v2(text, c, W, L), המקבלת בנוסף קוד האפמן c, ומחזירה את ייצוג הביניים המתאים.

שימו לב – בדומה למימוש המקורי, לכל תו בטקסט נמצא חזרה מקסימלית שמתחילה בתו זה. נקודד את החזרה באמצעות תת הרשימה $[m, k]$ אמ"ם אורך הקידוד הבינארי של החזרה **<u>קטן ממש</u>** מאורך הקידוד הבינארי תו-תו לפי קוד האפמן c.

<u>דוגמאות הרצה</u>: (שימו לב כי ערכי W, L שונים מהערכים הדיפולטיבים שראיתם בכיתה)

```
>>> c = {'a':'0', 'b':'10', 'c':'110', 'd':'1110', 'e':'1111'}
>>> LZW_compress_v2("abcdeabccde", c, 2**5-1, 2**3-1)
['a', 'b', 'c', 'd', 'e', 'a', 'b', 'c', [6, 3]]
>>> LZW_compress_v2("ededaaaaa", c, 2**5-1, 2**3-1)
['e', 'd', [2, 2], 'a', 'a', 'a', 'a', 'a']
```

.ii ממשו את הפונקציה inter_to_bin_v2(intermediate, c, W, L), המקבלת גם היא את קוד ההאפמן c (שאיתו ייצרו את ייצוג הביניים intermediate), ומחזירה את מחרוזת הביטים הדחוסה המתאימה לייצוג זה.

<u>דוגמת הרצה</u>:

```
>>> c = {'a':'0', 'b':'10', 'c':'110', 'd':'1110', 'e':'1111'}
>>> inter_to_bin_v2(['e', 'd', [2, 2]], c, 2**5-1, 2**3-1)
0111101110100010010  # indicator bits are colored in red for better readability
```

<u>רמז</u>: השינויים הדרושים הם קצרים ומקומיים, ומרבית הקוד נשאר זהה לקוד שראיתם בכיתה. אין צורך להסתבך. ניתן להיעזר בקוד שראיתם בכיתה.

<u>הערה</u>: השתכנעו שהפונקציה LZW_decompress הממירה את ייצוג הביניים לטקסט המקורי תעבוד ללא שינוי (ובפרט היא אינה תלויה בקוד האפמן c שאיתו קודדנו את המחרוזת).

.iii <u>הערה</u>: בסעיף זה נניח לשם פשטות כי $W = 2^5 - 1$ ו-$L = 2^3 - 1$.

בזכות תכונת ה-prefix free של קוד האפמן, האלגוריתם שמימשתם יכול לפענח כל מחרוזת בינארית דחוסה באופן יחיד. בפרט, מתקיימת הטענה הבאה:

לכל קוד האפמן c ולכל text1 != text2 הדחיסה תניב שתי מחרוזות בינאריות שונות.

וודאו שאתם מבינים מדוע טענה זו נכונה עבור קודים שהם prefix-free.

נרצה לבחון האם הטענה נכונה גם עבור קודים שאינם prefix-free.
לפניכם שני תנאים על הקוד c:

a. c הוא קוד חח״ע <u>שאינו</u> prefix-free (ובפרט אינו קוד האפמן) וגם <u>ואינו</u> uniquely decodable.
b. c הוא קוד חח״ע <u>שאינו</u> prefix-free (ובפרט אינו קוד האפמן) אבל <u>כן</u> uniquely decodable.

עבור כל אחד משני התנאים a ו-b (בנפרד), הוכיחו / הפריכו את הטענה הבאה. אם הטענה נכונה, הסבירו בקצרה כיצד לפענח ממחרוזת בינארית לייצוג הביניים המתאים, ואם היא לא נכונה תנו דוגמה נגדית.

<u>טענה</u>: לכל קוד c המקיים את התנאי, ולכל text1 != text2 הדחיסה מניבה שתי מחרוזות בינאריות שונות. כלומר, לכל c, text1, text2 כנ״ל מתקיים:

```
>>> bin1 = inter_to_bin_v2(LZW_compress_v2(text1, c), c)
>>> bin2 = inter_to_bin_v2(LZW_compress_v2(text2, c), c)
>>> bin1 != bin2
True
```

<u>סוף</u>