תרגיל בית מספר 5 - להגשה עד 2 ינואר בשעה 23:55

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

הגשה:

- תשובותיכם יוגשו בקובץ pdf ובקובץ py בהתאם להנחיות בכל שאלה.
- השתמשו בקובץ השלד skeleton5.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 hw5_012345678.pdf ו- hw5_012345678.py.
- מכיוון שניתן להגיש את התרגיל בזוגות, עליכם בנוסף למלא את המשתנה SUBMISSION_IDS שבתחילת קובץ השלד. רק אחת הסטודנטיות בזוג צריכה להגיש את התרגיל במודל.

- 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

השאלה עוסקת בעצים בינאריים, ובמחלקה Binary_search_tree. הניחו בשאלה זו שהמפתחות (השדה key) בצמתים הינם מחרוזות והערכים (השדה val) בצמתים הינם מספרים מטיפוס int גדולים ממש מ- 0 וקטנים או שווים ל- 100 .

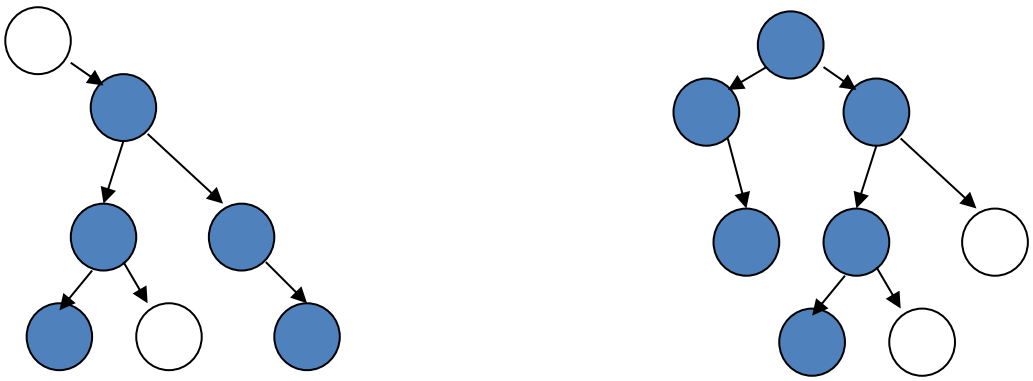
שימו לב שבכל הדוגמאות בסעיפים הבאים המפתחות הינם מטיפוס str.

א. קוטר של עץ הינו אורכו של מסלול ארוך ביותר בין שני צמתים בעץ, כאשר צומת לא מופיע במסלול יותר מפעם אחת, ואין חשיבות לכיוון הקשתות (ראו דוגמה בהמשך). אורך המסלול במקרה זה נקבע לפי מספר הצמתים במסלול. שימו לב שייתכנו מספר מסלולים שונים שאורכם הוא קוטר העץ. ממשו את המתודה `diam(self)` שמחזירה את קוטר העץ. עבור עץ ריק המתודה תחזיר 0.

<u>הנחיות מחייבות:</u>

1. על המתודה לפעול ב O(n) כאשר n מספר הצמתים בעץ.
2. אין להוסיף שדות ל Tree_node.

להלן שתי דוגמאות בהן מודגש בכחול מסלול שאורכו הינו הקוטר:

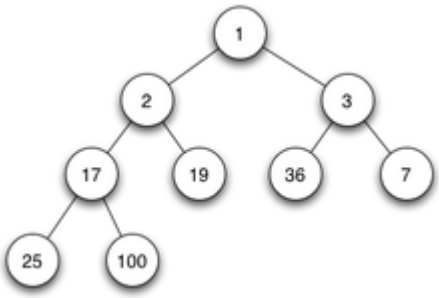


דוגמאות הרצה:

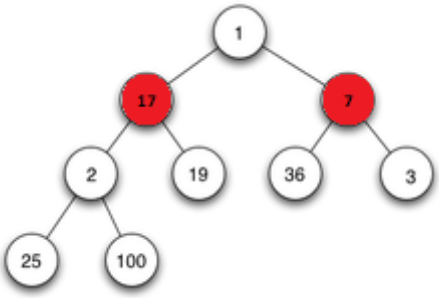
```
>>> t2 = Binary_search_tree()
>>> t2.insert('c', 10)
>>> t2.insert('a', 10)
>>> t2.insert('b', 10)
>>> t2.insert('g', 10)
>>> t2.insert('e', 10)
>>> t2.insert('d', 10)
>>> t2.insert('f', 10)
>>> t2.insert('h', 10)
>>> t2.diam()
6
>>> t3 = Binary_search_tree()
>>> t3.insert('c', 1)
>>> t3.insert('g', 3)
>>> t3.insert('e', 5)
>>> t3.insert('d', 7)
>>> t3.insert('f', 8)
>>> t3.insert('h', 6)
>>> t3.insert('z', 6)
>>> t3.diam()
5
```

ב. "ערימת מינימום" היא עץ בינארי "כמעט מושלם", כלומר כזה שכל השורות, פרט אולי לאחרונה, מלאות, והאחרונה מלאה משמאל עד לנקודה מסוימת. בנוסף, היא מקיימת: כל ערך (value) של צומת אינו גדול משני בניו

דוגמה לערימת מינימום בינארית:



לא ערימת מינימום:



a. בקובץ השלד ממשו את המתודה is_min_heap של Binary_search_tree שבודקת אם כל ה- values של הצמתים שלו (ולא ה keys) מקיימים את תכונת "ערימת המינימום". שימו לב שרק המפתחות רלוונטיים לסידור של המבנה כעץ חיפוש, ורק ה- values רלוונטיים לסידור המבנה כערימת מינימום.

הנחות והנחיות:

- על הפתרון להיות <u>רקורסיבי</u>
- אפשר להניח שהקלט תקין, כלומר עץ בינארי כמעט מושלם.
- הפונקציה תחזיר True אם העץ הוא ערימת מינימום ביחס ל- values שלו ו False אחרת.
- **אין** להשתמש בלולאות for, while

b. ציינו בקובץ ה pdf מהי סיבוכיות זמן הריצה עבור עץ בגודל n. הסבירו את ניתוח הסיבוכיות ונמקו את תשובתכם.

<u>הנחיות והבהרות לכלל השאלה</u>:

- בכל מקום בו נדרש לממש מתודה באופן רקורסיבי, מותר להגדיר פונקציית עזר רקורסיבית שתיקרא על ידי המתודה (שבמקרה זה תהיה "מתודת מעטפת").

רצוי ואף מומלץ לממש את פונקציית העזר הרקורסיבית כפונקציה פנימית של המתודה. לדוגמה:

```
def some_method(self):
    def helper_rec( <some parameters> ):
        # some code
    return helper_rec( <some arguments> )
```

- המחלקה Tree_node שראיתם בכיתה מופיעה בקובץ השלד. כאמור, אין לשנות מחלקה זו.

שאלה 2

בשאלה זו 3 חלקים בלתי תלויים

חלק א:

בחלק זה נממש תור עדיפות (priority queue) על בסיס רשימה מקושרת. למבנה זה יש שתי פעולות עיקריות:

insert(value, p) – מכניסה לתור צומת בעל ערך value ועדיפות p שהינה מספר שלם חיובי.

pull() – מוציאה מהתור את הצומת בעל העדיפות הגדולה ביותר ומחזירה את (value, p) שלו (כ Tuple). אם יש צמתים בעלי עדיפות זהה, יוחזר הצומת שנוסף ראשון לתור מבניהם. אם תור העדיפות ריק אז המתודה תחזיר (None,None) (כ Tuple).

צומת בתור עדיפות, שייוצג ע"י המחלקה PNode, מכיל ערך (value), מצביע לצומת הבא ברשימה (next), ועדיפות (priority). התור עצמו (מטיפוס PQueue) מכיל מצביע לצומת הראשון ברשימה (head) ואת אורך התור (len).

להלן מתודות האתחול של שתי המחלקות שהצגנו והמתודה len של המחלקה PQueue:

```
class PNode():

    def __init__(self, val, p, next=None):
        self.value = val
        self.next = next
        self.priority = p

class PQueue():

    def __init__(self, vals=None, ps=None):
        self.next = None
        self.len = 0
        if vals != None:
            for val, p in zip(vals, ps):
                self.insert(val, p)

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

בקובץ השלד מומשה עבורכם המתודה __repr__ של המחלקה PQueue ושל המחלקה PNode.
שימו לב – מתודת האתחול של PQueue עשויה להשתמש במתודה insert.
הנחיות:

- השלימו בקובץ השלד את המתודות insert ו pull של המחלקה PQueue לפי ההנחיות למעלה.
- על המתודה insert לפעול בזמן O(n) ובזיכרון נוסף O(1).
- על המתודה pull לפעול בזמן O(1) ובזיכרון נוסף O(1).

הערה: ניתן לממש תור עדיפות באופן יעיל יותר ע"י מבנה נתונים בשם ערימה (heap)

חלק ב:

נתון לכם מימוש מצומצם של linked_list בקובץ השלד. עליכם להשלים בו את המתודה reverse_start_end(self, k) אשר מקבלת מספר שלם אי-שלילי k אשר קטן או שווה לאורך הרשימה חלקי 2 והופכת את הסדר של k האיברים הראשונים והאחרונים ברשימה.

דוגמא: אם הרשימה lst היא 1->2->3->4->5->6->7->8 אז הפעולה lst.reverse_start_end(3) תעדכן את lst להיות 3->2->1->4->5->8->7->6

הנחיות:

- על המתודה לפעול בזמן O(n) ובזיכרון O(1) כאשר n הוא מספר האיברים ברשימה
- על המתודה לשנות את הרשימה עליה היא נקראת (כלומר – לא ליצור רשימה חדשה)
- אין צורך לבדוק את תקינות הקלט

חלק ג:

רשימת מזלג (ForkedList) היא רשימה אשר באחד מהצמתים שלה (לכל היותר) עשויה להתפצל לשתי תתי-רשימות.

כדי לממש זאת, צמתים ברשימה יהיו מהטיפוס Node2 אשר מכיל ערך (value), מצביע אחד לצומת ברשימה (next1), ומצביע שני לצומת ברשימה (next2). מהגדרת המבנה, יש לכל היותר צומת אחד ברשימה אשר בו שני המצביעים next1,next2 שונים מ None.

בקובץ ה pdf הציעו אלגוריתם שמקבל כקלט רשימת מזלג ומחזיר True אמ״ם יש מעגל ברשימה. את האלגוריתם ניתן לתאר ע״י הסבר מילולי או כפסאודו-קוד. אין צורך (אך אפשר) לממש בפייתון.

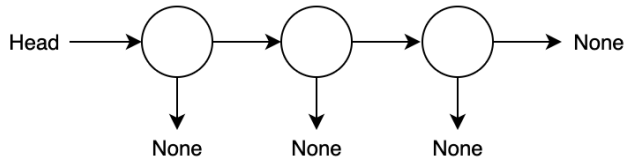
הנחיות:

- לשם פשטות התיאור שלכם, ניתן להניח ש next1 הוא None רק בקצוות הרשימה ובהתאם next2 שונה מ None רק בצומת הפיצול. כלומר, תמיד משתמשים במצביע next1 לפני שמשתמשים ב next2.
- הפונקציה צריכה לרוץ בסיבוכיות זמן O(n) ומקום נוסף O(1).

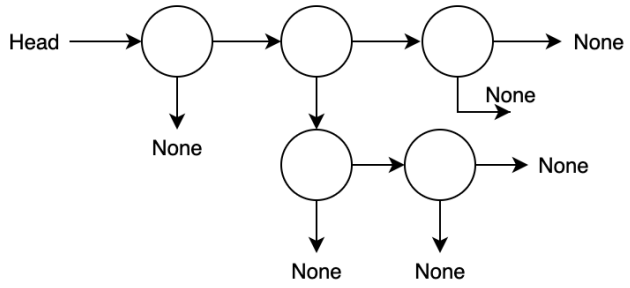
רמז: השתמשו ברעיון דומה למה שראיתם בתרגול.

דוגמאות בעמוד הבא:

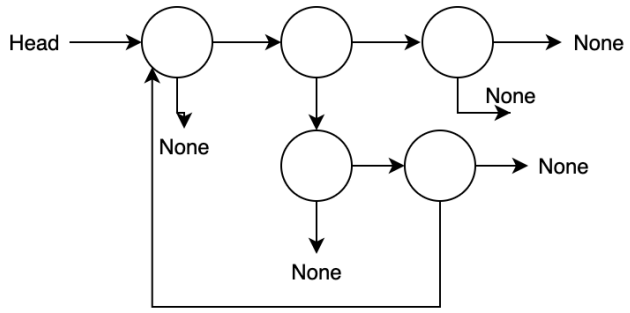
No Forks ForkedList



ForkedList with no cycle



ForkedList with cycle



שאלה 3

א. הוכיחו כי לכל $a, b, c \in \mathbb{N}$ מתקיים: $(a \bmod b)^c \bmod b = a^c \bmod b$

השתמשו בנוסחה הבאה, הידועה כנוסחת הבינום, עבור n טבעי:

$$(x+y)^n = \sum_{i=0}^{n} \frac{n!}{i!(n-i)!} x^i y^{n-i}$$

ב. בתרגול ראינו את הפונקציה crack_DH אשר בהינתן $g, p, x = g^a \bmod p$ מנסה למצוא a' כך ש- $g^{a'} \bmod p =$ $g^a \bmod p$. להלן תזכורת לקוד:

```
def crack_DH(p, g, x):
    ''' find secret "a" that satisfies g**a%p == x
        Not feasible for large p '''
    for a in range(1, p - 1):
        if a % 100000 == 0:
            print(a) #just to estimate running time
        if pow(g, a, p) == x:
            return a
    return None #should never get here
```

בנוסף, ראינו כי יתכן שהפונקציה תחזיר $a' \neq a$ עבורו מתקיים השיוויון לעיל.

הוכיחו כי בהינתן $g^b \bmod p,\ g^a \bmod p,\ p,\ g$ ו-a' כך ש- $g^{a'} \bmod p = g^a \bmod p$ ניתן לחשב ביעילות את $g^{ab} \bmod p$.

רמז – היעזרו בסעיף הקודם.

שאלה 4

א. בחלק זה נדון בסיבוכיות של פעולות חיבור, כפל והעלאה בחזקה של מספרים בכתיב בינארי.

חיבור וכפל של מספרים בינאריים יכול להתבצע בצורה דומה מאוד לזו של מספרים עשרוניים. להלן המחשה של אלגוריתם החיבור והכפל של שני מספרים בינאריים A,B:

```
  1 1 1 1     (carried digits)

      1 1 1 1 (A)
+   1 1 0 1 0 (B)
-------------
= 1 0 1 0 0 1
```

להלן המחשה של אלגוריתם ההכפלה AxB:

```
            1 0 1    (A)
        × 1 0 1 0    (B)
        ---------
            0 0 0    ← Corresponds to a zero in B
  +       1 0 1      ← Corresponds to a one in B
  +     0 0 0
  +   1 0 1
  ---------------
  =   1 1 0 0 1 0
```

בדוגמה הנ"ל יש 19 פעולות. 12 עבור הכפל והשאר עבור החיבור.

בהרצאה נלמד אלגוריתם iterated squaring להעלאה בחזקה של מספרים שלמים חיוביים (ללא מודולו). הקוד מופיע בהמשך. הניחו כי מספר הביטים בייצוג הבינארי של a הינו n ומספר הביטים בייצוג הבינארי של b הינו m.

עליכם לנתח את סיבוכיות זמן הריצה של הפעולה a^b בשיטה זו כתלות ב-m ו/או n ולתת חסם הדוק ככל האפשר במונחי $O(\cdot)$. שימו לב שהמספרים המוכפלים גדלים עם התקדמות האלגוריתם, ויש לקחת זאת בחשבון. כמו כן, בניתוח נתייחס רק לפעולות הכפל המופיעות באלגוריתם. אמנם באלגוריתם ישנן פעולות נוספות מלבד כפל. אך לפעולות אלו סיבוכיות זמן מסדר גודל זניח לעומת פעולות הכפל. למשל, ההשוואה b>0 רצה בזמן $O(1)$ (לא נכנסנו לעומק של ייצוג מספרים שלמים שליליים, אבל ראינו ברפרוף את שיטת המשלים ל- 2 ושם ברור שמספיק לקרוא את הביט השמאלי כדי להכריע האם מספר שלם הוא חיובי או שלילי). הפעולה b%2 דורשת בדיקת הביט הימני של b בלבד ולכן רצה בזמן $O(1)$. פעולת החילוק b//2 כוללת העתקה של b ללא הביט הימני ביותר למקום חדש בזיכרון, פעולה שגם כן רצה בזמן ליניארי בגודל של b. על ההסבר להיות תמציתי ומדוייק. מומלץ לחשוב על המקרה ה"גרוע" מבחינת מספר פעולות הכפל ולתאר כיצד הגעתם לחסם עבור פעולות אלו.

```
def power(a,b):
    """ computes a**b using iterated squaring
        assumes b is nonnegative integer """
    result = 1
    while b>0: # b is nonzero
        if b%2 == 1: # b is odd
            result = result*a
        a = a*a
        b = b//2
    return result
```

ב. בחלק זה תדרשו לממש וריאציות של פעולת החזקה בשיטת iterated squaring.

a. השלימו בקובץ השלד את הפונקציה power_new שבהינתן a, b מחשבת את a^b תוך ביצוע אותו מספר של פעולות כפל כמו הפונקציה power שלמעלה. יש להשלים 3 שורות בלבד.

```
def power_new(a,b):
    """ computes a**b using iterated squaring
        assumes b is nonnegative integer """
    result = 1
    b_bin = bin(b)[2:]
    reverse_b_bin = b_bin[::-1]
    for bit in reverse_b_bin:
    ____________________________________________
    ____________________________________________
    ____________________________________________
    return result
```

b. בסעיף הקודם ראינו שאלגוריתם iterated_squaring למעשה מתייחס ל b בייצוג בינארי. בסעיף זה נממש אלגוריתם iterated_squaring שלמעשה מתייחס ל b בבסיס base שהוא מספר שלם גדול או שווה ל 2. השלימו בקובץ השלד את הפונקציה power_with_base שבהינתן a, b, base מחשבת את a^b. תוכן הפונקציה נמצא בקובץ השלד בהערה. עליכם להסיר את הפקודה pass שבראשית הפונקציה, להסיר את סימני ההערות מהקוד ולהשלים את השורות החסרות בלבד.

```
def power_with_base(a, b, base=2):
    """ computes a**b using iterated squaring
        with divisor base >= 2
        assumes b is nonnegative integer """
    result = 1
    while b>0:
        x = 1
        residual = ___________________________
        for i in range(residual):
            x *= a
        ____________________________________________
        for i in range(___________________):
            x *= a
        ____________________________________________
        b = b // base
    return result
```

שאלה 5

נתונה רשימה של n מחרוזות $[s_0, s_1, \dots, s_{n-1}]$, לאו דווקא שונות זו מזו. בנוסף נתון k חיובי, 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

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 לסיפא של s1, ויש חפיפה באורך k בין הרישא של s1 לסיפא של s2. שימו לב שאנו לא מתעניינים בחפיפות אפשריות של מחרוזות עם עצמן, כמו למשל החפיפה באורך 5 בין רישא של s0 לסיפא שלה 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(…). הניחו כמובן 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(56, "a")
>>> d.insert(56, "b")
>>> d #calls __repr__
0 []
1 []
2 [[56, 'a'], [56, 'b']]

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

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 אותה 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(…). 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.

ו. לסיום נרצה להשתמש במילון של פייתון (טיפוס dict), כדי לפתור את אותה הבעיה שוב. שימו לב להבדל מהפתרון הקודם: כאן אין לנו אפשרות לשנות את המימוש הפנימי של המחלקה dict, ובפרט איננו יכולים להחליט שחזרות של מפתחות מותרות (כזכור במילון של פייתון אין חזרות של מפתחות). השלימו את הפונקציה prefix_suffix_overlap_hash2(lst, k) בהתאם למגבלה זו.

ז. בצעו השוואה של זמני ריצה בין שלושת הפתרונות, עבור רשימת מחרוזות באורכים של לפחות 1000 תווים, וצרפו את מסקנותיכם **כולל הסברים** (בקצרה) על סיבת ההבדלים בזמני הריצה. אין צורך לצרף את זמני הריצה עצמם.

<u>סוף</u>