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

תרגיל בית מספר 5 - להגשה עד 27/12/2021 בשעה 23:55

קראו 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.

- 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.

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

שאלה 1

א. היזכרו בהגדרה של $a \ mod \ b$:

נאמר ש- $a \ mod \ b = r$ אם ניתן לרשום את a בתור $a = k \cdot b + r$ עבור $k \in \mathbb{Z}$, או באופן שקול אם ניתן לרשום את r בתור $r = t \cdot b + a$ עבור $t \in \mathbb{Z}$ (בפרט מתקיים הקשר $t = -k$).

הוכיחו את התכונות החשובות הבאות:

i. לכל $a, b, c \in \mathbb{N}$ מתקיים $\big((a \ mod \ b) + (c \ mod \ b)\big) mod \ b = (a + c) mod \ b$.

ii. לכל $a, b, c \in \mathbb{N}$ מתקיים $\big((a \ mod \ b) \cdot (c \ mod \ b)\big) mod \ b = (a \cdot c) mod \ b$.

iii. הסיקו מ-ii באינדוקציה כי לכל $a, b, c \in \mathbb{N}$ מתקיים: $(a \ mod \ b)^c \ mod \ b = a^c \ mod \ b$.

ב. ראינו בכיתה כי בפרוטוקול *Diffie-Helman*, בהינתן g, p המפתחות הפומביים, ו-$x = g^a \ mod \ p$ המסר שמחושב על ידי אליס ונשלח באופן גלוי, לא ידועה דרך יעילה למצוא את המפתח הסודי a (בעיה זו נקראת גם בעיית הלוג הדיסקרטי). להלן קטע קוד ממחברת תרגול 8 אשר מנסה בהינתן g, p, x למצוא את המפתח הסודי a על ידי מעבר על פני כל ערכי a האפשריים:

```
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 pow(g, a, p) == x:
            return a
    return None #should never get here
```

שימו לב כי יתכן שהפונקציה תחזיר $a' \neq a$ עבורו מתקיים השוויון לעיל.

למשל, עבור הפרמטרים $p = 7, g = 13$ והמפתח הסודי $a = 4$, נקבל כי $x = 13^4 \ mod \ 7 = 1$ אבל מתקיים כי $13^2 \ mod \ 7 = 1$ ולכן האלגוריתם הנ"ל יחזיר את $a' = 2 \neq a$. נרצה להוכיח שגם במקרה כזה מציאת a' תעזור לנו "לפצח" את הפרוטוקול ולגלות את הסוד המשותף:

הוכיחו כי בהינתן המפתחות הפומביים g, p וגם y,x כך ש: $x = g^a \ mod \ p; \ y = g^b \ mod \ p$. אם נדע לבחור a' כך ש- $g^a \ mod \ p = g^{a'} \ mod \ p$ ניתן לחשב ביעילות את הסוד המשותף $g^{ab} \ mod \ p$. רמז – היעזרו בסעיף א'.

שאלה 2

בהינתן מספר טבעי $n > 0$, **הגורמים הראשוניים שמרכיבים את n** הם רשימת המספרים הראשוניים שמכפלתם שווה ל-n. כלומר, אם נסמן ב-P את רשימת הגורמים הראשוניים של n (כולל חזרות) באורך k. מתקיים:

$$\prod_{0 \le i < k} P[i] = n$$

וכן כל $p \in P$ הוא מספר ראשוני.

לדוגמה, אם $n = 12$ אז $P = [2, 2, 3]$ שכן $2 \cdot 2 \cdot 3 = 12$, וכל האיברים ב-P הם ראשוניים. שימו לב שאותו גורם ראשוני יכול להופיע מספר פעמים.

בשאלה זו נממש מספר מתודות במחלקה FactoredInteger אשר מייצגת מספרים טבעיים באמצעות רשימה ממוינת בסדר עולה של הגורמים הראשוניים שלהם. ניתן להיווכח שייצוג זה הוא **ייצוג טוב** – כלומר שיש התאמה חח״ע ועל בין קבוצת המספרים הטבעיים לבין קבוצת כל הסדרות העולות הסופיות של מספרים ראשוניים (רשות: הוכיחו זאת).

העשרה:

- בעיית הפירוק לגורמים ראשוניים (דהיינו: בהינתן מספר n, מצאו את רשימת הגורמים הראשוניים שלו) היא בעיה קשה שלא ידוע לה אלגוריתם בעל זמן ריצה פולינומי (ב-log(n) שהוא אורך הקלט במקרה שלנו). בדומה לבעיית הלוג הדיסקרטי עליה דיברנו בכיתה, על קושי זה מסתמכים רבים מפרוטוקולי ההצפנה המודרניים. עובדה מעניינת היא שקיים **אלגוריתם קוונטי** יעיל המפרק מספר לגורמיו הראשוניים. עובדה זו היא אחת הסיבות לכך שפיתוח מחשב קוונטי חזק הפכה למטרה מרכזית במדעי המחשב בשנים האחרונות. אתם מוזמנים לקרוא עוד על מחשב קוונטי ועל אלגוריתם שור בוויקיפדיה.

א. להלן מתודת האתחול של המחלקה FactoredInteger:

```
class FactoredInteger:

    def __init__(self, factors, verify=True):
        """ Represents an integer by its prime factorization """
        if verify:
            assert is_sorted(factors)
        number = 1
        for p in factors:
            if verify:
                assert(is_prime(p))
            number *= p
        self.number = number
        self.factors = factors
```

שימו לב שהמתודות is_sorted ו- is_prime (שאותה ראינו בתרגול) נתונות לכם בקובץ השלד. הפונקציה is_sorted בודקת האם רשימת הקלט ממויינת ומחזירה True אם כן ו-False אחרת. כמו כן, שימו לב שבאתחול של המחלקה FactoredInteger יש פרמטר בוליאני נוסף בשם verify שקובע האם לבדוק או

לא לבדוק את תקינות הקלט באתחול. באופן טבעי, הפרמטר הזה מוגדר להיות True באופן דיפולטי. מצד שני, יש מצבים שבוודאות אנחנו יודעים שיש לנו רשימה ממויינת של מספרים ראשוניים ובמצב כזה נוכל בעזרת הפרמטר לדלג על הבדיקות ולחסוך זמן.

נסמן ב-k את אורך הרשימה factors, וב-n את מספר הביטים בייצוג הבינארי של המספר number שמתקבל בסוף המתודה.

i. בהינתן n כלשהו, מהו טווח הערכים האפשריים של k בהנחה שהרשימה מאותחלת באופן תקין? הסבירו.

ii. הראו שסיבוכיות הזמן של פונקציית האתחול היא $O(n^3)$ במקרה הגרוע ביותר, ותנו דוגמה שמוכיחה שחסם זה הוא הדוק, כלומר לכל n תנו דוגמה לקלט שעבורו זמן הריצה של הפונקציה הוא $\Theta(n^3)$.

רמז: עבור רשימת הראשוניים $P = [p_1, \dots, p_k]$ סמנו את הרשימה המתאימה $A = [a_1, \dots, a_k]$ כאשר a_i מייצג את כמות הביטים במספר p_i. נתחו את סיבוכיות הפונקציה כתלות בערכי a_i. שימו לב שהתוצאה הסופית צריכה להיות תלויה רק ב n.

בסעיפים הבאים נממש מספר פונקציות המטפלות באובייקטים מסוג FactoredInteger. דרישות הסיבוכיות מוגדרות כתלות ב-k ו-m, שהם אורכי הרשימות factors של הפרמטרים self ו-other בהתאמה. בפרט, הניחו שפעולות אריתמטיות על האיברים בתוך factors ועל number נעשות בזמן $O(1)$.

ב. ממשו את הפונקציות המובנות הבאות של המחלקה FactoredInteger בקובץ השלד. שימו לב כי self ו-other הם אובייקטים מטיפוס FactoredInteger.

- **`__repr__(self)`** - מחזירה מחרוזת המייצגת את המספר באופן הבא:

  $$< number: p_1 * p_2, \dots * p_k >$$

  שימו לב שבמחרוזת אין כלל רווחים. לדוגמה, עבור המספר 12 הפונקציה מחזירה:

  `<12:2*2*3>`

- **`__eq__(self, other)`** – מחזירה True אם self ו-other מייצגים את אותו מספר, ו-False אחרת. הפונקציה צריכה לרוץ בזמן $O(1)$.

- **`__mul__(self, other)`** – מתודה מובנית שתומכת באופרטור *
  מחזירה אובייקט מסוג FactoredInteger המייצג את תוצאת המכפלה בין self ו-other.
  הפונקציה צריכה לרוץ בסיבוכיות זמן $O(k+m)$.

- **`__pow__(self, other)`** – מתודה מובנית שתומכת באופרטור **
  מחזירה אובייקט מסוג FactoredInteger המייצג את תוצאת החזקה $self^{other}$ הפונקציה צריכה לרוץ בסיבוכיות זמן $O(k * other)$. שימו לב שאין להשתמש באופרטור * של המחלקה.

<u>דוגמאות הרצה</u>:

```
>>> n1 = FactoredInteger([2, 3])       # n1.number = 6
>>> n2 = FactoredInteger([2, 5])       # n2.number = 10
>>> n3 = FactoredInteger([2, 2, 3, 5]) # n3.number = 60

>>> n3                                 # __repr__
<60:2*2*3*5>
>>> n1 == FactoredInteger([2,3])       # __eq__
True
>>> n1 * n2                            # __mult__
<60:2*2*3*5>
>>> n1 ** n2                           # __pow__
<60466176:2*2*2*2*2*2*2*2*2*2*3*3*3*3*3*3*3*3*3*3>
```

ג. היזכרו בהגדרה של gcd שראיתם בכיתה: בהינתן $n, k \in \mathbb{N}$, **המחלק הגדול ביותר** של n ו-k (הנקרא גם **gcd** – greatest common divider), הוא המספר המקסימלי אשר מחלק גם את n וגם את k. לדוגמה, ה-gcd של 12 ו-8 הוא 4 שכן 4 מחלק גם את 8 וגם את 12, וכל מספר גדול מ-4 לא מחלק את שניהם. ממשו את הפונקציה **`gcd(self, other)`** המחזירה אובייקט מטיפוס FactoredInteger המייצג את המחלק הגדול ביותר של self ו-other. הפונקציה צריכה לרוץ בסיבוכיות זמן של $O(k+m)$. שימו לב שאלגוריתם אוקלידס שראיתם בכיתה לא מקיים זאת.

ד. בהינתן $n, k \in \mathbb{N}$, ***הכפולה המשותפת המינימלית*** (הנקראת גם **lcm** – least common multiple) היא המספר הקטן ביותר שמתחלק גם ב-n וגם ב-k. לדוגמה, הכפולה המשותפת המינימלית של 6 ו-15 היא 30 שכן 30 מתחלק גם ב-6 וגם ב-15, וכל מספר קטן מ-30 לא מתחלק בשניהם (זהו ה"מכנה המשותף" המינימלי, כפי שאתם מכירים מחיבור שברים).

i. בקובץ השלד נתון לכם מימוש של הפונקציה **`lcm(self, others)`**. הסבירו בקצרה מה עושה הפונקציה ואיך היא מחשבת את ה-lcm של self וכל ה-FactoredInteger ברשימה **`others`**.

ii. נניח שסה"כ משתתפים בחישוב m מספרים ראשוניים (סה"כ כל אורכי הרשימות בכל האובייקטים שמשתתפים נסכמים ל-m). נתחו את סיבוכיות זמן הריצה של הפונקציה על בסיס m.

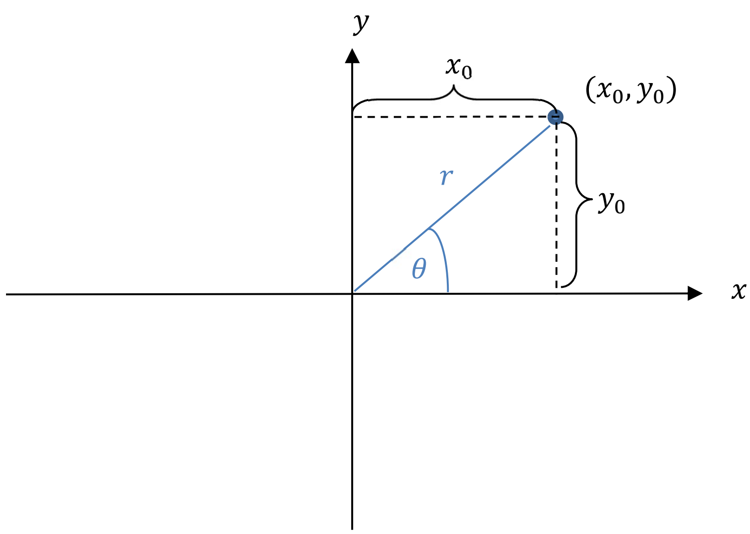
דוגמאות הרצה:

```
>>> n1 = FactoredInteger([2, 2, 3])   # n1.number = 12
>>> n2 = FactoredInteger([2, 2, 2])   # n2.number = 8
>>> n1.gcd(n2)
<4:2*2>
>>> n3 = FactoredInteger([2, 3])
>>> n4 = FactoredInteger([3, 5])
>>> n3.lcm([n4,n2])
<60:2*2*3*5>
```

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

<u>שאלה 3</u>

בשאלה זו נעסוק בנקודות במישור, כלומר נקודות שניתן למקם על מערכת צריכים דו-מימדית. כל נקודה (x_0, y_0) ניתנת לייצוג על ידי **קואורדינטות פולריות** $(r(x_0, y_0), \theta(x_0, y_0))$, כאשר $r \in \mathbb{R}^+$ הוא המרחק של הנקודה (x_0, y_0) מראשית הצירים, ו- $\theta \in [0,2\pi)$ היא הזווית ברדיאנים בין ציר ה-x לנקודה (x_0, y_0). ראו שרטוט להמחשה:



נגדיר את המחלקה Point אשר תייצג נקודה במישור. נרצה לשמור גם את הקואורדינטות הסטנדרטיות (הנקראות **קואורדינטות קרטזיות**) וגם את הקואורדינטות הפולריות. להלן מתודת האתחול של המחלקה:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.r = math.sqrt(x**2 + y**2)
        self.theta = math.atan2(y, x)
        if self.theta < 0:
            self.theta = angle + 2*math.pi
```

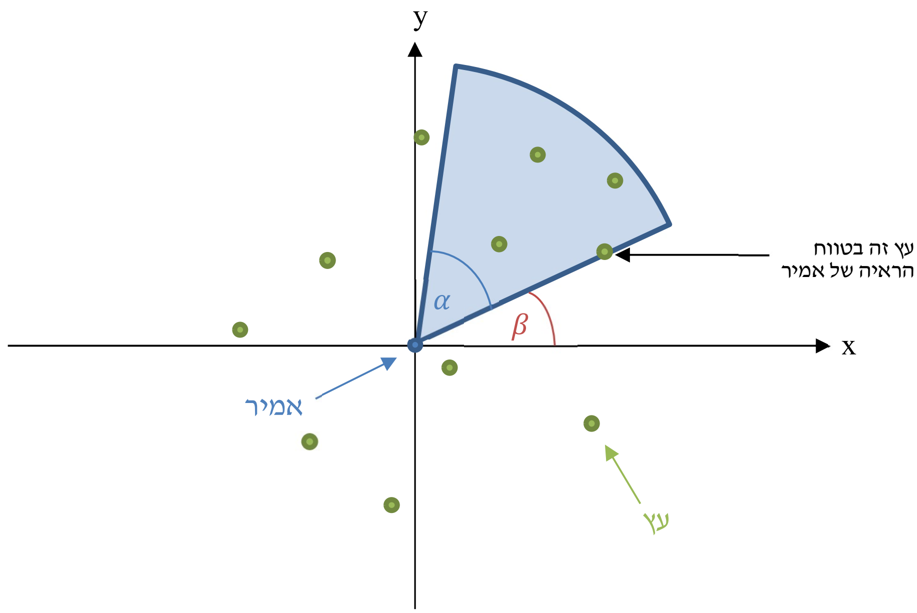
<u>הערה</u>: atan2 היא פונקציה מספריית math שמחשבת את הזווית הרצויה בטווח בין $-\pi$ ל-π, אבל ב-Point self.theta תמיד תהיה בטווח $[0,2\pi)$.

<u>הערות כלליות לשאלה</u>:

- עקרונית לרוב לא נרצה לאפשר למשתמש חיצוני לגשת אל השדות הפנימיים של המחלקה. עם זאת, לשמירה על פשטות המחלקה, בתרגיל זה ניתן לגשת לשדות של Point באופן ישיר.
- השדות במחלקה הם מטיפוס float. כזכור, חישובים אריתמטיים ב-float הם לא מדויקים מטבעם. בפרט, בשאלה זו ניתן להתעלם משגיאות הנובעות מחוסר דיוק של float.

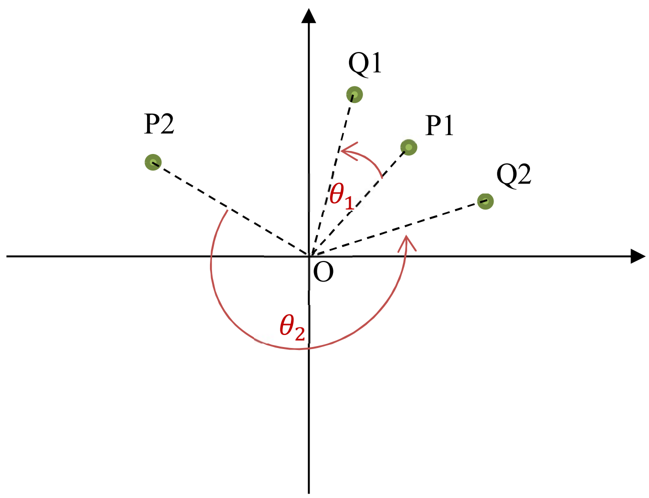
בשאלה זו נרצה להשתמש במחלקה Point על מנת לעזור לאמיר בפתרון שתי בעיות.

א. אמיר עומד ביער, ומסביבו n עצים. בהינתן זווית ראיה (זאת אומרת זווית בעלת אורך אינסופי) α, עליה למצוא את הזווית β אליו הוא צריך להסתכל על מנת ללכוד בזווית הראיה שלו כמה שיותר עצים.
תחילה נרצה למדל את השאלה באמצעות מונחים איתם יהיה לנו קל יותר לעבוד: נייצג את העצים באמצעות נקודות במישור, כאשר $(0,0)$ היא הנקודה בה עומד אמיר, במטרה לדמות "מבט על" של היער. ראו שרטוט להמחשה:



עלינו למצוא את הזווית β שתמקסם את מספר הנקודות הירוקות בתוך משולש הפיצה הכחול (כולל שפת המשולש). הניחו כי אף עץ חלילה לא מוחץ את אמיר – כלומר אין עץ בנקודה $(0,0)$. נפתור את השאלה בשני שלבים:

i. נסמן ב-O את ראשית הצירים. ממשו את הפונקציה angle_between_points(self, other) במחלקה Point, המקבלת 2 נקודות (אובייקטים מסוג Point), נסמנן $P = self, Q = other$, ומחזירה את הזווית בה יש לסובב את הקטע PO נגד כיוון השעון כדי להגיע לקטע QO. הפונקציה מחזירה זווית בטווח $[0, 2\pi)$. לפניכם שרטוט של שני זוגות נקודות, והזוויות המתאימות להן:



דוגמאות הרצה:

```
>>> p1 = Point(1, 1)             # p1.theta = 0.25 * pi
>>> p2 = Point(0, 3)             # p2.theta = 0.5 * pi
>>> p1.angle_between_points(p2) # self = p1, other = p2
0.78539816339                    # 0.25 * pi
>>> p2.angle_between_points(p1) # self = p2, other = p1
5.49778714378                    # 1.75 * pi
```

ii. ממשו את הפונקציה **find_optimal_angle(trees, alpha)** המקבלת רשימה trees של n נקודות (אובייקטים מטיפוס Point), וזווית ראיה $\text{alpha} \in (0, 2\pi)$, ומחזירה את הזווית $\beta \in [0,2\pi)$ הממקסמת את מספר העצים שרואה אמיר (תיתכן יותר מזווית β אחת מתאימה – החזירו זווית כלשהי).

<u>הנחיה</u>: על הפונקציה לרוץ בסיבוכיות זמן $O(nlogn)$ כאשר n מספר העצים.

<u>רמז</u>: התחילו בלמיין את רשימת הנקודות על פי ערכי ה-θ שלהן והיעזרו בפונקציה שמימשתם בסעיף הקודם.

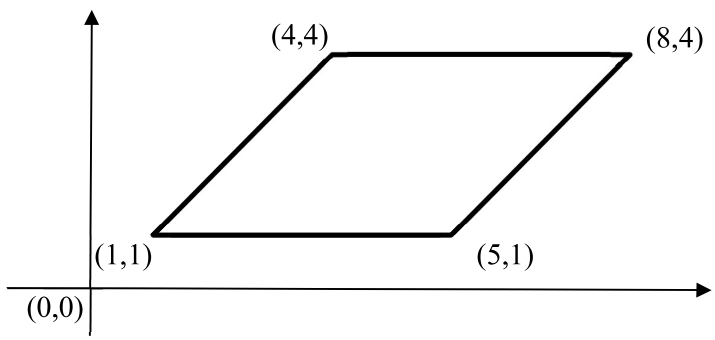
דוגמת הרצה (מומלץ לשרטט את הנקודות על מערכת צירים כדי להבין מדוע זה ערך ההחזרה):

```
>>> trees = [Point(2,1),Point(0,3),Point(-1,3),
Point(-1,1),Point(-1,-1),Point(0,-5)]
>>> find_optimal_angle(trees, 0.25 * math.pi)
1.5707963267948966  # 0.5 * pi
```

השתכנעו שבדוגמה זו יש זווית **אחת ויחידה** שעונה על השאלה. כאמור, בדוגמאות אחרות יתכן כי יותר מזווית אחת מתאימה (בפרט, יתכנו אינסוף זוויות מתאימות).

ב. בגאומטריה, מצולע (באנגלית, *Polygon*) הוא חלק ממישור המתוחם על ידי מספר סופי של קטעים. כל אחד מהקטעים הללו נקרא צלע וכל נקודה בה נפגשות שתי צלעות סמוכות נקראת קודקוד. במקרה שלנו, נרצה למדל מצולע על ידי רשימה מקושרת של אובייקטים מסוג *Point*. זאת אומרת, המצולע מוגדר על ידי רשימה מקושרת של קודקודים ככה שכל שני קודקודים סמוכים ברשימה מחוברים עם צלע וכן הקודקוד הראשון והקודקוד האחרון ברשימה מחוברים גם הם עם צלע (למרות שהם לא מחוברים ברשימה). שימו לב שהמחלקה *Polygon* שנמצאת בקובץ השלד.
בנוסף, נרצה גם להשתמש במחלקה *Segment* (קטע סגור) שמייצגת קו ע״י 2 נקודות. המחלקה נמצאת בקובץ השלד.

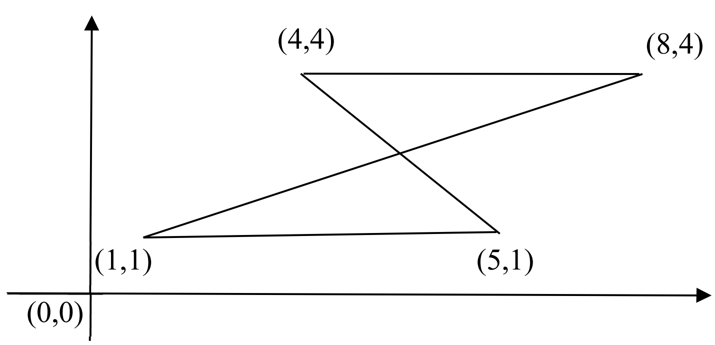
לדוגמא המקבילית הבאה :



תאותחל על ידי הפקודה הבאה :

```
>>> parallelogram = 
Polygon(Linked_list([Point(1,1),Point(4,4),Point(8,4),
                     Point(5,1)]))
```

- בשאלה זו לשם הפשטות נניח כי כל המצולעות שלנו נמצאים ברביע הראשון, זאת אומרת שכל ערכי ה-x וכל ערכי ה-y גדולים שווים ל-0.
- שימו לב, כי סדר הנקודות משנה. זאת אומרת שסדר שונה של נקודות קובע מצולע אחר.
- מצולע פשוט הוא מצולע שצלעותיו נחתכות אחד על ידי השניה בקצותיהן בלבד. שימו לב שהמצולע הבא אינו מצולע פשוט :



והוא מאותחל על ידי הפקודה הבאה (סדר הנקודות שונה לעומת המקבילית) :

```
>>> not_simple =
Polygon(Linked_list([Point(1,1),Point(8,4),Point(4,4),
                     Point(5,1)]))
```

i. ממשו את המתודה edges(self) של המחלקה Polygon שמחזירה רשימה (list של פייתון) של הזווית הפנימית (הזוויות שנמצאות בתוך המישור התחום על ידי המצולע) במעלות (כלומר מספר בין 0 ל-360) לכל קודקוד במצולע. זאת אומרת, המקום ה-i של הרשימה המוחזרת תהיה הזווית שמתאימה לקודקוד ה-i ברשימה המקושרת של הפוליגון עם השכן שלו מימין והשכן שלו משמאל.

הערות:

- הזוויות של נקודה במצולע מחושבת על ידי הזווית בין שני הקטעים שמתחילים בנקודה הזו. זאת אומרת, שתצטרכו לחשב את הזווית בין שלשות של נקודות. הקוד לחישוב הזווית נמצא בקובץ השלד.
- שימו לב שבסעיף זה אנחנו רוצים להשתמש במעלות ולא ברדיאנים כמו בסעיף א. זאת כדי להקל עליכם, מכיוון שבמעלות יהיה לכם קל יותר להבין מציור מה בערך הזווית בכל קודקוד של המצולע.
- אתם יכולים להניח שמדובר במצולע פשוט.
- אל תשכחו את הקצוות של הרשימה.
- על המתודה לרוץ בסיבוכיות זמן $O(n)$.

לדוגמא:

```
>>> parallelogram.edges()
[45.0, 135.0, 45.0, 135.0]
```

ii. ממשו את המתודה simple(self) של המחלקה *Polygon*, שבודקת האם הוא מצולע פשוט).

הערות:

- אתם יכולים להשתמש במחלקה *Segment* ובפרט בפונקציה *intersecting*.
- על המתודה לרוץ בסיבוכיות זמן $O(n^2)$.
- אל תשכחו את הקצוות של הרשימה.

לדוגמא:

```
>>> parallelogram.simple()
True
>>> not_simple.simple()
False
```

<u>שאלה 4</u>

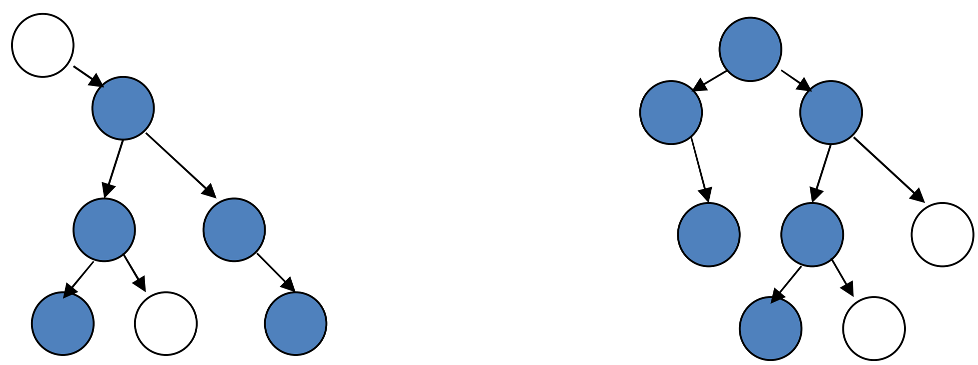
השאלה עוסקת בעצי חיפוש בינאריים, ובמחלקה Binary_search_tree. הניחו בשאלה זו שהמפתחות (השדה key) בצמתים הינם מחרוזות והערכים (השדה val) בצמתים הינם מספרים מטיפוס int גדולים ממש מ-0. שימו לב שבכל הדוגמאות בסעיפים הבאים המפתחות הינם מטיפוס str.

א. **קוטר של עץ** מוגדר להיות 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</u>:

1. על המתודה לרוץ בסיבוכיות זמן $O(n)$ כאשר n מספר הצמתים בעץ.
2. אין להוסיף שדות ל- Tree_node.

להלן שתי דוגמאות בהן מודגש בכחול מסלול שאורכו הינו הקוטר. שימו לב שצעד במסלול יכול להיות מצומת אל אחד מבניו וגם מצומת אל אביו.

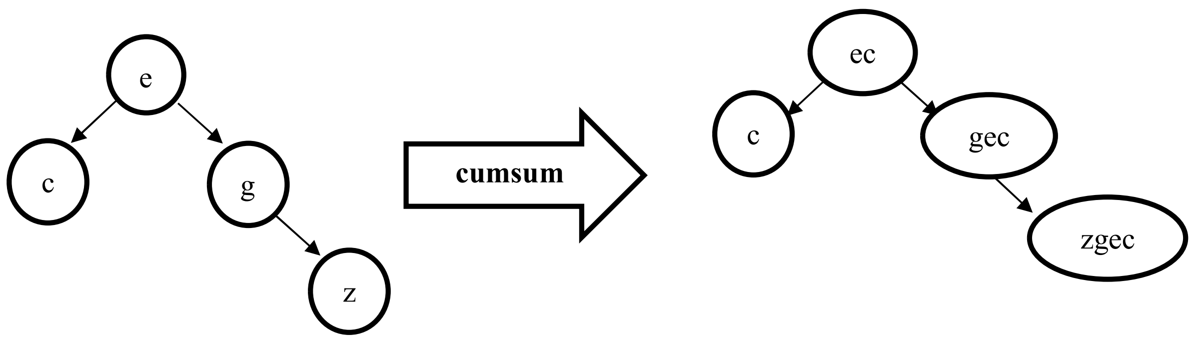


דוגמאות הרצה:

```
>>> 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. נגדיר פעולת cumsum קיצור של (cumulative sum) המחליפה את המפתח של כל צומת בעץ בשרשור של כל ההמפתחות הקטנים או שווים לו לקסיקוגרפית. לדוגמא:



לדוגמא: בדוגמא זו למשל המפתח *e* בשורש מוחלף במחרוזת *ec*: שרשור המפתח המקורי בשורש (דהיינו *e*) וכל המפתחות הקטנים ממנו (דהיינו *c*). המפתח *c* לא משתנה כי אין מפתח קטן ממנו.
המפתח *g* בצומת הימנית מהשורש מוחלף במחרוזת *gec*: המפתח המקורי בצומת זה (דהיינו *g*) ושרשור כל המפתחות הקטנים ממנו (*e,c*). בצומת הימנית ביותר בעץ המפתח *z* מוחלף במחרוזת *zgec*: המפתח המקורי בצומת זה (דהיינו *z*) ושרשור כל המפתחות הקטנים ממנו (*g,e,c*).

i. השלימו את המתודה cumsum(self) של המחלקה Binary_search_tree שמבצעת את פעולת העדכון המתוארת.

הנחיות:

- כמו הרבה מהמתודות שראינו במימוש של עץ חיפוש בינארי, יש לממש פתרון רקורסיבי.
- ניתן לממש את הפתרון באמצעות פונקציית מעטפת.
- המתודה cumsum אינה מחזירה עץ חדש אלא עושה את העדכון על העץ עצמו.

ii. הסבירו בקצרה מדוע העץ לאחר ביצוע הפעולה cumsum הוא עדיין עץ חיפוש בינארי.

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

שאלה 5 – 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 הרישא (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 מחרוזות באורך k נמקו את תשובתכם בקצרה.

<u>סוף</u>