אוניברסיטת תל אביב - בית הספר למדעי המחשב
מבוא מורחב למדעי המחשב, סתיו 2017

תרגיל בית מספר 5 - להגשה עד 8 בינואר (יום ראשון) 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.
- 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, סתיו 2017

שאלה 1

בקובץ השלד תמצאו מימוש חלקי למחלקה Binary המייצגת מספר טבעי (גדול או שווה 0) בכתיב בינארי. המספר ייוצג כמחרוזת.
לשם פשטות, לאורך כל השאלה נניח כי אין אפסים מובילים במספרים הבינאריים, למעט המספר "0". אין צורך לבדוק זאת במתודות שתכתבו או להתייחס למקרה כזה.

א. ממשו את המתודות הבאות למחלקה:

- `__lt__` – מימוש האופרטור > המשווה בין שני מספרים בינאריים של המחלקה.
- `__add__`– מימוש האופרטור + המאפשר חיבור של שני מספרים בינאריים מהמחלקה.
- `is_power_of_two` - בדיקה האם המספר הבינארי המיוצג הוא חזקה של 2.
- `largest_power_of_two` - החזרת החזקה הכי גבוהה של 2 שהינה קטנה או שווה למספר הבינארי בבסיס דצימלי

<u>הערה:</u> אין להשתמש בפונקציות bin ו- int של פייתון במימוש `__lt__` ו- `__add__`

ב. אמירג רוצה לדעת מה שארית החלוקה של מספר בינארי ב- 3. הוא התחיל לממש את המתודה div3, שאמורה להחזיר את שארית החלוקה ב- 3 של מספר מהמחלקה Binary. השלימו את שתי השורות החסרות בפונקציה div3 בקובץ השלד. עליכן להשתמש אך ורק במתודות שמימשתן בסעיף הקודם.

ג. מיכל מעוניינת לחשב שארית מודולו 3 של מספרים בינאריים בני אלפי ביטים, וחוששת שהמתודה מהסעיף הקודם לא תסתיים עד תחילת סמסטר האביב.
היא שמה לב לתכונות הבאות:

- כאשר מוסיפים למספר בינארי 0 מימין (לדוגמה, משנים "1" ל "10"), אם שארית החלוקה שלו ב- 3 לפני ההוספה הייתה 1, אז שארית החלוקה שלו ב 3 אחרי ההוספה תהיה 2.
- כאשר מוסיפים למספר בינארי 1 מימין (לדוגמה, משנים "1" ל "11"), אם שארית החלוקה שלו ב- 3 לפני ההוספה הייתה 1, אז שארית החלוקה שלו ב 3 אחרי ההוספה תהיה 0.

מיכל שמה לב כי באופן דומה, ניתן לדעת מה תהיה שארית החלוקה ב 3 של מספר בינארי לאחר הוספת 0 או 1 מצד ימין, גם כאשר שארית החלוקה שלו **לפני ההוספה** הייתה 0 או 2.

השלימו בקובץ השלד את המימוש של div3_new (שמחזירה אותם ערכים כמו div3).

ד. ציינו והסבירו מהי סיבוכיות הזמן של המתודות div3_new ,div3 כתלות במספר הביטים *n* של self. תנו תשובה הדוקה ככל שתוכלו במונחי O(…).

דוגמאות הרצה:

```
>>> a = Binary("101")
>>> b = Binary("10")
>>> a < b
False
>>> a + b
0b111
>>> a.div3()
2
```

```
>>> b.is_power_of_two()
True
>>> a.largest_power_of_two()
4
```

שאלה 2

בהרצאה ראינו מימוש למבנה הנתונים "עץ חיפוש בינארי". במימוש שראינו, צומת בעץ יוצג ע"י אובייקט מהמחלקה Tree_node, ואילו העץ עצמו **לא** יוצג בגישת OOP.

א. ממשו בקובץ השלד מחלקה בשם BSearch_Tree עבור עץ חיפוש בינארי. המחלקה תכיל שדה יחיד בשם root המייצג את שורש העץ. שורש העץ, כמו כל יתר הצמתים, יהיה אובייקט מטיפוס Tree_node . בעת האיתחול של אובייקט עץ חיפוש בינארי הוא יהיה ריק מצמתים.

ב. ממשו באופן רקורסיבי את המתודות הבאות עבור המחלקה BSearch_Tree :

a. insert(self, key, val) – הכנסת צומת חדש בעל מפתח key וערך val (כזכור, סידור הצמתים בעץ מתייחס למפתח key). אם כבר קיים צומת בעל מפתח key, לא יוכנס צומת חדש אלא יעודכן <u>הערך</u> של הצומת הקיים להיות val.

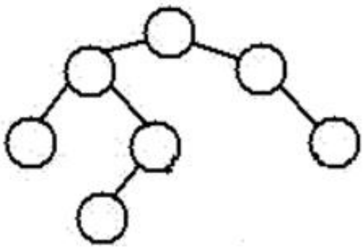
b. lookup(self, key) – החזרת הערך (val) של צומת בעל מפתח key. אם לא קיים יוחזר None.

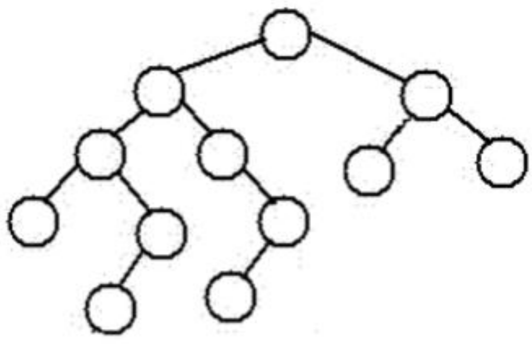
c. sum(self) – החזרת סכום המפתחות של צמתי העץ. הניחו בשאלה זו כי המפתחות בעץ הם מטיפוס int.

d. find_closest_key(self, search_key) - החזרת <u>המפתח</u> (key) של צומת בעץ שקרוב ביותר ל search_key מבין כל המפתחות בעץ. אם יש כמה מפתחות שקרובים באותה מידה ל search_key, יוחזר אחד מהם, לבחירתכם. הניחו בשאלה זו כי המפתחות בעץ הם מטיפוס int.

ג. הגדרה: עץ בינארי הוא מאוזן אם מתקיימים שני התנאים הבאים:
1. תת-העץ השמאלי שלו וגם תת-העץ הימני שלו הם עצים מאוזנים
2. ההפרש בין עומקו של תת-העץ השמאלי לבין עומקו של תת-העץ הימני הוא לכל היותר 1.

לדוגמה, בתמונה שלמטה העץ השמאלי הוא מאוזן, אבל העץ הימני אינו מאוזן, כי ההפרש בין עומק תת-העץ השמאלי של השורש לבין עומק תת-העץ הימני של השורש הוא 2.





ממשו באופן רקורסיבי את המתודה is_balanced(self) אשר מחזירה True אם העץ מאוזן, ו False אחרת.
<u>הנחיה מחייבת</u>:

1. על המתודה לפעול ב O(n) כאשר n מספר הצמתים בעץ.

2. אין להוסיף שדות ל Tree_node.

<u>הנחיות והבהרות לכלל השאלה</u>:

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

דוגמאות הרצה:

```
>>> bin_tree = BSearch_tree()
>>> bin_tree.insert(2,"hi")
>>> bin_tree.insert(4,"bye")
>>> bin_tree.insert(1,"hello")
>>> bin_tree.insert(3,"lovely")
>>> bin_tree.sum() == 10
True
>>> bin_tree.lookup(3) == "lovely"
True
>>> bin_tree.lookup(100) == None
True
>>> bin_tree.find_closest_key(5) == 4
True
>>> bin_tree.is_balanced()
True
>>> bin_tree.insert(5,"dear")
>>> bin_tree.insert(6,"tea")
>>> bin_tree.is_balanced()
False
```

שאלה 3

בשאלה זו עליכם להגדיר מחלקה חדשה בשם Polynomial, שתייצג פולינום במשתנה אחד. להזכירכם, פולינום במשתנה אחד מדרגה n ניתן לכתוב באופן הבא: $p(x) = a_nx^n + a_{n-1}x^{n-1} + \cdots + a_1x + a_0$ כאשר $n \geq 0$ מספר טבעי, ו- $a_n \neq 0$, למעט בפולינום האפס $p(x) = 0$.

המתודות __init__ ו-__repr__ כבר ממומשות עבורכם בקובץ השלד. המקדמים של הפולינום נשמרים בשדה **coeffs** שהינו רשימה (list), שבה האיבר במקום ה- i מייצג את המקדם של x^i. שימו לב כי המקדם האחרון חייב להיות שונה מ- 0, כדי למנוע מצב שבו פולינום מיוצג ע"י רשימה ארוכה מהמינימום הדרוש. בכל הסעיפים, לצורך ניתוח סיבוכיות הניחו כי חיבור וכפל של שני מספרים כלשהם רץ בזמן קבוע O(1).

א. השלימו בקובץ השלד את מימוש המתודות הבאות. בכל מתודה מצויינת מהי סיבוכיות זמן הריצה הדרושה - לקבלת ניקוד מלא יש לעמוד בדרישות סיבוכיות אלו. ניתן להניח כי הקלט תקין. היעזרו בדוגמאות ההרצה שמופיעות בהמשך. (שימו לב שבקובץ השלד מופיעה המתודה __lt__ אותה אין צורך לממש!)

- **degree** – מחזירה את הדרגה של הפולינום. סיבוכיות זמן דרושה O(1). הניחו כי הפונקציה len של מחלקת הרשימות (list) רצה בזמן קבוע O(1).
- **evaluate** – מקבלת כקלט מספר x (שלם או ממשי) ומחזירה את הערך של הפולינום עבור ה-x הנתון. סיבוכיות זמן דרושה O(n).

  <u>הערה מחייבת</u>: במתודה זו אין להשתמש כלל בפעולת העלאה בחזקה (pow או **). יש לחשב את התוצאה ע"י פעולות כפל וחיבור בלבד.
- **derivative** – מחזירה אובייקט <u>חדש</u> מטיפוס Polynomial שהוא הנגזרת של הפולינום המקורי. סיבוכיות זמן דרושה O(n). פעולת הנגזרת של פולינום מוגדרת כדלקמן:

$$derivative(a_nx^n + a_{n-1}x^{n-1} + \cdots + a_1x + a_0) = n \cdot a_nx^{n-1} + (n-1) \cdot a_{n-1}x^{n-2} + \cdots + a_1$$

- **__eq__** – מקבלת כפרמטר אובייקט נוסף other מטיפוס Polynomial, ומחזירה True אם הוא מייצג אותו פולינום כמו self, אחרת False. סיבוכיות הזמן הדרושה היא $O(1)$ כאשר שני הפולינומים לא מאותה מעלה ו- $O(n)$ אם שני הפולינומים מדרגה n.
- **__add__** – מקבלת כפרמטר אובייקט נוסף other מטיפוס Polynomial, ומחזירה אובייקט <u>חדש</u> מטיפוס Polynomial, שמייצג את סכום שני הפולינומים. אם דרגת הפולינום הקיים (self) היא n ודרגת הפולינום הנוסף היא m אז סיבוכיות הזמן הדרושה היא O($max\{m,n\}$).
- **__mul__** – מקבלת כפרמטר אובייקט נוסף other מטיפוס Polynomial, ומחזירה אובייקט <u>חדש</u> מטיפוס Polynomial, שמייצג את מכפלת שני הפולינומים. אם דרגת הפולינום הקיים (self) היא n ודרגת הפולינום הנוסף היא m אז סיבוכיות הזמן הדרושה היא O(mn).
- **__neg__** – מחזירה אובייקט <u>חדש</u> מטיפוס Polynomial שהוא הפולינום שמתקבל ע"י מכפלת כל אחד מהמקדמים של self ב (-1). ראו את דוגמת ההרצה בהמשך. סיבוכיות זמן דרושה O(n).

- **`__sub__`** – מקבלת כפרמטר אובייקט נוסף other מטיפוס Polynomial, ומחזירה אובייקט חדש מטיפוס Polynomial, שמייצג את תוצאת החיסור של הפולינום other מהפולינום הקיים (self), כלומר את הפולינום self-other. אם דרגת הפולינום הקיים (self) היא n ודרגת הפולינום הנוסף היא m אז סיבוכיות הזמן הדרושה היא $O(max\{m,n\})$. יש להשתמש במימוש של **`__neg__`**.

דוגמאות הרצה:

```
>>> q = Polynomial([0, 0, 0, 6])
>>> q
(6*x^3)
>>> Polynomial([0, 0, 0, -6])
(-6*x^3)
>>> q.degree()
3
>>> p = Polynomial([3, -4, 1])
>>> p
(3*x^0) + (-4*x^1) + (1*x^2)
>>> p.evaluate(10)
63
>>> dp = p.derivative()
>>> dp
(-4*x^0) + (2*x^1)
>>> ddp = p.derivative().derivative()
>>> ddp
(2*x^0)
>>> q==p
False
>>> r = p+q
>>> r
(3*x^0) + (-4*x^1) + (1*x^2) + (6*x^3)
>>> p + Polynomial([-1])
(2*x^0) + (-4*x^1) + (1*x^2)
>>> q == Polynomial([0, 0, 0, 5]) + Polynomial([0, 0, 0, 1])
True
>>> -p
(-3*x^0) + (4*x^1) + (-1*x^2)
>>> p-q
(3*x^0) + (-4*x^1) + (1*x^2) + (-6*x^3)
>>> p*q
(18*x^3) + (-24*x^4) + (6*x^5)
>>> Polynomial([0])*p
0
```

ב. השלימו את מימוש המתודה find_root ששייכת למחלקה Polynomial. על המתודה להשתמש בשיטת ניוטון רפסון (NR) למציאת קירוב לשורש כלשהו של הפולינום. שימו לב שאם לפולינום יותר משורש ממשי אחד, ייתכן כי בהרצות חוזרות יוחזר שורש שונה (בגלל הניחוש ההתחלתי האקראי של NR). במתודה find_root יש להשלים שורה אחת בודדת (באפשרותכם להשתמש בנגזרת סימבולית או נומרית, לבחירתכם).

דוגמת הרצה:

```
>>> p.find_root()
0.9999999996240886
>>> p.find_root()
3.000000000000003
```

<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 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 סיפא (סיומת) של s_i לרישא (התחלה) של s_j. כלומר s_i[-k:] == s_j[:k].

לדוגמה, אם האוסף מכיל את המחרוזות

s_0 = "a"*10

s_1 = "a"*6 + "b"*4

s_2 = "a" + "b"*4 + "c"*5

אז עבור k=5 יש חפיפה באורך k בין הסיפא של s_0 לרישא של s_1, ויש חפיפה באורך k בין הסיפא של s_1 לרישא של s_2. שימו לב שאנו לא מתעניינים בחפיפות אפשריות של מחרוזות עם עצמן, כמו למשל החפיפה באורך 5 בין סיפא של s_0 לרישא 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 s_0 = s_1 = "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 הסיפא באורך k שלה אל מול כל הרישות באורך k של כל המחרוזות האחרות. ממשו את הפתרון הזה בקובץ השלד, בפונקציה suffix_prefix_overlap(lst, k), אשר מקבלת רשימה (מסוג list של פייתון) של מחרוזות, וערך מספרי k, ומחזירה רשימה עם כל זוגות האינדקסים (tuples) של מחרוזות שיש ביניהן חפיפה כנ״ל. אין חשיבות לסדר הזוגות ברשימה, אך יש כמובן חשיבות לסדר הפנימי של האינדקסים בכל זוג.

דוגמאות הרצה:

```
>>> s1 = "a"*10
>>> s2 = "a"*6 + "b"*4
>>> s3 = "a" + "b"*4 + "c"*5
>>> suffix_prefix_overlap([s1,s2,s3], 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 סיפות באורך k של המחרוזות הנתונות, ואילו הערך שנלווה לכל סיפא כזו הוא האינדקס של המחרוזת ממנה הגיעה הסיפא (מספר בין 0 ל- n-1). חישוב ה- hash לצורך הכנסה וחיפוש במילון מתבצע על המפתח בלבד. 2) מכיוון שיכולות להיות סיפות 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 suffix_prefix_overlap_hash1(lst, k), שהגדרתה זהה לזו של suffix_prefix_overlap(lst, k), אלא שהיא תשתמש במחלקה Dict מהסעיף הקודם. כאמור, כל הסיפות יוכנסו למילון תחילה, ואז נעבור על כל הרישות ונבדוק לכל אחת אם היא נמצאת במילון.

ה. ציינו מהי סיבוכיות הזמן של הפתרון הזה **בממוצע**, כתלות ב- n וב- k במונחים של O(…). הניחו כמובן כי השוואה בין שתי תת מחרוזות באורך k דורשת $O(k)$ פעולות במקרה הגרוע, וכך גם חישוב hash על מחרוזת באורך k.

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 suffix_prefix_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 ההבדלים בזמני הריצה.

<u>שאלה 5</u>

להלן הצעה לטיפול בהתנגשויות בטבלאות hash: במקום הרשימות ה"פנימיות", נשמור טבלאות hash "פנימיות". כלומר בכל כניסה של הטבלה הראשית ישנה רשימה של רשימות. בנוסף, נצטרך להגדיר שתי פונקציות hash במקום אחת: $h_1(x) = x \ mod \ m_1$ תמפה מפתחות לכניסות בטבלה הראשית שגודלה m_1, ואילו $h_2(x) = x \ mod \ m_2$ שתמפה מפתחות לכניסות בטבלאות ה"פנימיות", שגודלן m_2.

נגדיר "התנגשות חיצונית" בין שני מפתחות $x \ != y$ כמצב בו מתקיים $h_1(x) = h_1(y)$, ו"התנגשות פנימית" כמצב בו מתקיים $h_1(x) = h_1(y) \ and \ h_2(x) = h_2(y)$.

ענו על כל השאלות להלן בקובץ ה pdf.

א. נניח כי הטבלה הראשית היא בגודל $m_1=5$ וכל טבלה פנימית היא בגודל $m_2=3$. מכניסים לטבלה את המפתחות 0, 5, 12, 15. האם תתרחש התנגשות פנימית בין שני מפתחות כלשהם? אם כן, ציינו מיהם המפתחות שיתנגשו ומדוע, אחרת הסבירו מדוע לא.

ב. מהי סיבוכיות החיפוש <u>במקרה הגרוע</u> של מפתח במימוש הנ"ל של טבלת hash כתלות במספר האיברים n שנמצאים בטבלה? תנו תשובה כחסם אסימפטוטי הדוק ככל שתוכלו ונמקו.

ג. עבור טבלה ראשית בגודל $m_1=5$ וכל טבלה פנימית בגודל $m_2=3$, מכניסים לטבלה 5 מספרים שונים זה מזה כלשהם. לאחר מכן מחפשים בטבלה את המספר 2, וידוע שהתקבל המקרה הגרוע של זמן החיפוש. רישמו דוגמה אפשרית ל- 5 המספרים שהוכנסו לטבלה טרם החיפוש.

ד. אמירג הציע לבחור את גודל הטבלה הראשית להיות חזקה שלמה של גודל כל טבלה פנימית, כלומר אם גודל טבלה פנימית הוא m_2 אז גודל הטבלה הראשית יהיה $m_1 = m_2^k$ עבור $k \geq 2$ שלם כלשהו. מהו החיסרון העיקרי בבחירת פרמטרים כזו? הסבירו.

שאלה 6

משולש פסקל הוא משולש אינסופי המורכב משורות של מספרים, כך שבשורה ה- n, האיבר ה- r מוגדר כ:

$$a_{nr} = \binom{n}{r} = \frac{n!}{r!(n-r!)}.$$

7 השורות הראשונות במשולש הן כדלהלן:

						1						
					1		1					
				1		2		1				
			1		3		3		1			
		1		4		6		4		1		
	1		5		10		10		5		1	
1		6		15		20		15		6		1

נשים לב שניתן לקבל את השורה ה-i במשולש פסקל מהשורה ה-i-1 באופן הבא:

1. האיבר הראשון והאיבר האחרון בשורה הם תמיד 1.
2. פרט לכך, האיבר ה-j בשורה ה-i הוא סכום האיבר ה-j-1 והאיבר ה-j בשורה ה-i-1.

נייצג כל שורה במשולש על ידי רשימה כך שהשורה ה-0 היא הרשימה [1], השורה ה-1 היא הרשימה[1,1] וכו׳.

<u>סעיף א</u>

השלימו בקובץ השלד את הפונקציה `next_row(row)`, המקבלת את השורה ה-i-1 במשולש פסקל ומחזירה (return) את השורה ה-i.

<u>סעיף ב</u>

נייצג את משולש פסקל כזרם אינסופי של רשימות בצורה הבאה:

`[[1], [1,1], [1,2,1], [1,3,3,1],…`

השלימו בקובץ השלד את פונקצית הגנרטור `generate_pascal()` אשר מחזירה גנרטור המייצר את משולש פסקל. כלומר, בכל קריאה ל- next על הגנרטור להחזיר את השורה הבאה במשולש פסקל - בצורת רשימה.

דוגמת הרצה:

```
>>> gen = generate_pascal()
>>> next(gen), next(gen), next(gen)
([1], [1,1], [1,2,1])
```

<u>סעיף ג</u>

משולש ברנולי הוא המשולש שבו כל שורה היא רשימת הסכומים החלקיים של המקדמים הבינומיים,

כלומר $a_{nk} = \sum_{i=0}^{k} \binom{n}{i}$.

במילים אחרות, כל שורה במשולש ברנולי היא רשימת הסכומים החלקיים של השורה המתאימה במשולש פסקל.
בפרט, המספר ה- i בשורה ה- j של משולש ברנולי הוא סכום i המספרים הראשונים של השורה ה-j במשולש פסקל.

7 השורות הראשונות במשולש ברנולי הן כדלהלן:

```
                  1
               1     2
            1     3     4
         1     4     7     8
      1     5    11    15    16
   1     6    16    26    31    32
1     7    22    42    57    63    64
```

נייצג את משולש ברנולי בצורה הדומה למשולש פסקל, כזרם אינסופי של רשימות:

```
[[1], [1,2], [1,3,4], [1,4,7,8] …
```

השלימו בקובץ השלד את פונקצית הגנרטור `generate_bernoulli()` אשר מחזירה גנרטור המייצר את משולש ברנולי.
דוגמת הרצה:

```
>>> gen = generate_ bernoulli()
>>> next(gen), next(gen), next(gen)
([1], [1,2], [1,3,4])
```

<u>הנחיות הגשה:</u>
ממשו את הפונקציות מסעיפים א׳, ב׳ ו ג׳ בקובץ השלד.

<u>סוף</u>