אוניברסיטת תל אביב - בית הספר למדעי המחשב
מבוא מורחב למדעי המחשב, חורף 2015

תרגיל בית מספר 4 (מעודכן) - להגשה עד 29 בדצמבר (יום שני) בשעה 23:55

שימו לב: שאלות 2 ו-5 עודכנו. השינויים מודגשים בצהוב

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 skeleton4.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 012345678.pdf ו- 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

מיכל ואמיר החליפו ביניהם מפתח סודי באמצעות פרוטוקול Diffie-Hellman, כפי שניתן לראות בהרצות הבאות (הפונקציות find_prime ו- DH_exchange הן אלו שראינו בתרגול):

```
>>> p = find_prime(10)
>>> p
593
>>> g, a, b, x, y, key = DH_exchange(p)
>>> g, a, b, x, y, key
(9, 530, 147, 574, 527, 156)
```

סטודנטית מהקורס מבוא מורחב מנסה לגלות את המפתח המשותף הנ"ל. לשם כך היא מאזינה לתשדורת בין מיכל ואמיר, ולכן יש ברשותה את x ואת y ששלחו זה לזו. בנוסף, כמובן, יש ברשותה את p ואת g הפומביים. כעת, מנסה הסטודנטית להריץ את הפונקציה crack_DH מהתרגול, שמטרתה לפתור את בעיית ה- Discrete log.

להלן ההרצה שביצעה:

```
>>> crack_DH(p,g,x)
234
```

כפי שניתן לראות, הפונקציה לא גילתה את הסוד המקורי a=530, אלא ערך אחר, a'=234. אך כמובן שהסטודנטית הזדונית איננה יודעת זאת.

האם יכולה הסטודנטית, באמצעות המידע שברשותה כעת, לחשב את הסוד המשותף של מיכל ואמיר (156 במקרה זה)? אם לדעתכם כן, הוכיחו זאת באופן מתמטי, עבור a'≠a כלשהו, ועבור p, g, x, y, b כלשהם. אם לדעתכם לא, הסבירו מדוע לא.

שאלה 2

בשאלה זו ננתח ונשווה את הסיבוכיות של מספר פונקציות רקורסיביות לחישוב מקסימום ברשימה.

את שתי הגרסאות הבאות, max1 ו max2, פגשנו בתרגול 6, ואף ניתחנו את סיבוכיות הזמן ואת עומק הרקורסיה שלהן:

```
def max1(L):
  if len(L)==1:
       return L[0]
   return max(L[0], max1(L[1:]))

def max2(L):
    if len(L)==1:
        return L[0]
    l = max2(L[:len(L)//2])
    r = max2(L[len(L)//2:])
    return max(l,r)
```

א. להלן הפונקציה max11, שהיא גרסה הדומה ל- max1, אך הפעם ללא slicing. פונקציה זו מקבלת בנוסף לרשימה L שני אינדקסים אשר תוחמים את אזור החיפוש הרלבנטי. הקריאה הרקורסיבית הראשונה מתבצעת מתוך פונקצית המעטפת max_list11:

```
def max11(L,left,right):
    if left==right:
        return L[left]
    return max(L[left], max11(L,left+1,right))

def max_list11(L):
    return max11(L,0,len(L)-1)
```

השימוש במנגנון כזה של פונקצית מעטפת מקובל מאוד, ומאפשר למשתמש להעביר לפונקציה אך ורק את הקלט הדרוש (הרשימה עצמה), ולא פרמטרים נוספים שקשורים לאופן מימוש רקורסיבי כזה או אחר (גבולות שמאלי וימני).

נתחו את סיבוכיות הזמן ואת עומק הרקורסיה של max11. על הניתוח לכלול:

1. ציור סכמטי של עץ הקריאות הרקורסיביות עבור רשימה באורך n. העץ יכיל צומת עבור כל קריאה רקורסיבית. בתוך הצומת רישמו את אורך הרשימה עליה בוצעה הקריאה, ולצד הצומת רישמו חסם אסימפטוטי הדוק במונחי $O(\ldots)$ על סיבוכיות הזמן של אותו צומת כפונקציה של n.
2. ציון עומק עץ הרקורסיה כפונקציה של n.
3. ציון סיבוכיות הזמן (חסם אסימפטוטי הדוק במונחי $O(\ldots)$) כפונקציה של n.

ב. השלימו בקובץ השלד את הפונקציה max22, שתעבוד באותו אופן כמו max2 (כלומר תבצע חלוקה של הבעיה בדיוק לאותן תתי בעיות), אבל הפעם ללא slicing. הקריאה הראשונה ל- max22 תהיה מתוך max_list22, כפי שמופיע בקובץ השלד:

```
def max_list22(L):
    return max22(L,0,len(L)-1)
```

ג. נתחו את סיבוכיות הזמן ואת עומק הרקורסיה של max22 בדומה לניתוח שביצעתם בסעיף א' 1-3 עבור max11.

ד. השלימו את הטבלה הבאה, המציינת את זמני הריצה של ארבע הפונקציות max1, max2, max_list11, max_list22, עבור רשימות בגודל n=1000, 2000, 4000. את זמני הריצה מדדו באמצעות מנגנון ה"סטופר" שהוצג בתרגיל בית 1, או באמצעות הפונקציה elapsed מההרצאות. זכרו שכדי לקבל תוצאות אמינות עדיף להריץ מספר גדול של הרצות ולמצע. מאחר שכברירת מחדל עומק הרקורסיה המקסימלי הינו 1000, חלק מהפונקציות לא יצליחו לרוץ. לכן יהיה עליכם לשנות את עומק הרקורסיה המקסימלי לנניח 5000 באמצעות הפקודה sys.setrecursionlimit(5000) (עקבו אחר ההנחיות בשקפים של הרצאה 13).

Function	n = 1000	n = 2000	n = 4000
max1			
max2			
max_list11			
max_list22			

ה. הסבירו **<u>בקצרה</u>** כיצד תוצאות המדידה מסעיף ד' מתיישבות עם סיבוכיות זמני הריצה מסעיפים א3, ג3 ומהתרגול (של max1 ו- max2). התייחסו לקצב הגידול של זמני הריצה כתלות בגודל הקלט.

שאלה 3

בתרגול ראינו את בעיית העודף ואת פתרונה הרקורסיבי. השלימו בקובץ השלד את הפונקציה change_fast, שתפתור גם היא את בעיית העודף באופן דומה, אך הפעם עם מנגנון של memoization לחיסכון בזמן ריצה.
הפונקציה, כמו קודמתה ללא memoization , מקבלת שני פרמטרים – amount ו- coins.

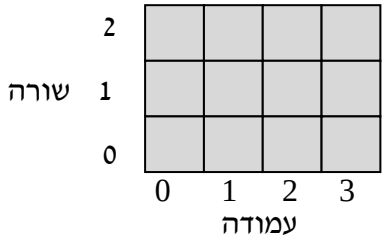
שאלה 4

בתרגול הוצג המשחק זלול! (Munch!), משחק לוח לשני שחקנים אשר גורעים (זוללים) בזה אחר זה קוביות מתוך טבלת שוקולד על פי חוקים מוסכמים מראש. זהו, בין השאר, גם משחק עם מסר חינוכי אקטואלי: הזלילה אינה מומלצת! השחקן המפסיד הוא זה אשר נאלץ לבלוע את קובית השוקולד האחרונה (השמאלית התחתונה).
כמו כן טענו שניתן להוכיח שקיימת "אסטרטגית ניצחון" עבור השחקן הפותח. פירוש הדבר הוא שאם השחקן הפותח (זה שמשחק ראשון) משחק היטב, מובטח כי ינצח, ללא תלות במהלכי המשחק של השחקן היריב. יש לציין שההוכחה מבטיחה קיום אסטרטגית ניצחון כזו, אך לא מהי (למשל מהו הצעד הראשון שעל השחקן הפותח לבצע על מנת לזכות).
<u>הערה</u>: השקפים הרלוונטיים שהוצגו בתרגול מופיעים באתר בעמוד התרגולים.

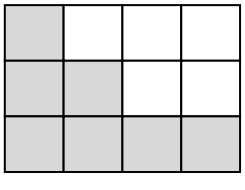
בשאלה זו נממש פונקציה אשר מקבלת מצב לוח חוקי ברגע נתון של המשחק (קונפיגורציה, ע"פ המינוח מהתרגול), ובודקת האם עבור השחקן אשר תורו לשחק כעת קיימת אסטרטגית ניצחון מקונפיגורציה זו (כלומר האם ינצח אם ישחק היטב מכאן והלאה). הפונקציה תחזיר True או False: האם קיימת או לא קיימת אסטרטגית ניצחון לשחקן הנוכחי, בהתאמה.

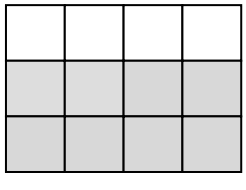
לדוגמה, עבור לוח במצב ההתחלתי, הפונקציה תחזיר True (כי כפי שטענו קודם ניתן להוכיח שלשחקן הפותח קיימת אסטרטגית ניצחון בראשית המשחק), ועבור לוח שנותרה בו רק הקובייה האחרונה הפונקציה תחזיר False (כי השחקן שתורו לשחק חייב לבלוע את הקובייה המורעלת הזו).

מצב הלוח הנוכחי מועבר לפונקציה בייצוג קומפקטי. ייצוג זה הוא רשימה באורך m שבאינדקס ה i שלה מופיע הגובה הנוכחי של הטור ה i בטבלה. שימו לב שעבור מצב לוח חוקי זו חייבת להיות סדרה מונוטונית יורדת (אבל לא יורדת ממש, כלומר ייתכנו אברים עוקבים בעלי ערך זהה).
לדוגמה, הייצוג הקומפקטי של מצב המשחק ההתחלתי של לוח בגודל m=4, n=3 הוא [3,3,3,3], וכך נראה הלוח:



בשלב מתקדם יותר של המשחק, עשוי הלוח להראות כך (הקוביות שצבועות בלבן הן אלו שכבר נגרעו), ולקבל את הייצוגים הבאים:


[3,2,1,1]


[2,2,2,2]

א. ממשו את הפונקציה שחתימתה היא win(hlst) אשר מקבלת כקלט את הייצוג הקומפקטי של מצב הלוח הנוכחי (hlst), ומחזירה True אם לשחקן שתורו לשחק כעת יש אסטרטגית ניצחון, ו False אם אין לו אסטרטגית ניצחון.
על הפונקציה להיות רקורסיבית, או פונקצית מעטפת של פונקציה רקורסיבית (ראו דוגמה לפונקצית מעטפת בשאלה 2 של תרגיל זה).
בסעיף זה אין להשתמש ב memoization.
הריצו את הפונקציה על קלטים שונים. בדקו ורשמו מהו ערך n הגדול ביותר עבורו הקריאה הבאה לפונקציה שכתבתם:

win([n]*(n+2)+[n-1])

(כלומר קריאה על טבלה מלאה בגודל n-by-(n+3) אשר הפינה הימנית העליונה שלה כורסמה) מחזירה True או False תוך דקה לכל היותר על המחשב שלכם.

ב. שפרו את הפונקציה מסעיף א כך שתכלול memoization. ממשו את השכלול בפונקציה שחתימתה win_fast(hlst). בדקו שוב ורשמו מהו ערך n הגדול ביותר עבורו הקריאה הבאה לפונקציה שכתבתם:

win_fast([n]*(n+2)+[n-1])

מחזירה True או False תוך דקה לכל היותר על המחשב שלכם.

שאלה 5

בשאלה זו עליכם להגדיר מחלקה חדשה בשם Polynomial, שתייצג פולינום במשתנה אחד. להזכירכם, פולינום במשתנה אחד מדרגה n ניתן לכתוב באופן הבא: $p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ 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).

הערה מחייבת: במתודה זו אין להשתמש כלל בפעולת העלאה בחזקה (pow או **). יש לחשב את התוצאה ע"י פעולות כפל וחיבור בלבד.

- **derivative** – מחזירה אובייקט חדש מטיפוס Polynomial שהוא הנגזרת של הפולינום המקורי. סיבוכיות זמן דרושה O(n). פעולת הנגזרת של פולינום מוגדרת כדלקמן:

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

- **__eq__** – מקבלת כפרמטר אובייקט נוסף other מטיפוס Polynomial, ומחזירה True אם הוא מייצג אותו פולינום כמו self, אחרת False. אם דרגת הפולינום הקיים (self) היא n ודרגת הפולינום הנוסף היא m אז סיבוכיות הזמן הדרושה היא O($max\{m,n\}$).
- **__add__** – מקבלת כפרמטר אובייקט נוסף other מטיפוס Polynomial, ומחזירה אובייקט חדש 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 חדש 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).
- **__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 $O(max\{m,n\})$.

- **__neg__** – מחזירה אובייקט חדש 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).

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.
דוגמת הרצה:

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

ג. פולינום **דליל** הינו פולינום שבו מספר המקדמים a_i השונים מ-0 (נסמנו ב- k) קטן מאוד ביחס ל- n ($k \ll n$).
לדוגמא: עבור $p(x) = 3x^{1000000} + 20x$ מתקיים k=2, n=1000000.

עליכם לממש את המחלקה Polynomial מחדש. המימוש החדש יהיה בעל ייצוג פנימי שונה (כלומר שדות פנימיים שונים), ויאפשר למתודה derivative לרוץ בסיבוכיות זמן $O(k)$. כלומר עבור פולינום דליל המתודה תהיה יעילה יותר אסימפטוטית מאשר בסעיף א'. השתדלו ששאר המתודות יהיו גם כן יעילות ופשוטות במימוש החדש. יחד עם זאת לא יורדו נקודות על אי יעילות של מתודות אלו. השלימו את המימוש החדש במחלקה Polynomial_sparse. יש לממש מחדש את כל המתודות שמומשו קודם, וגם את __init__ ואת __repr__.

<u>הנחיה מחייבת</u>: המשתמש במחלקה יוכל לאתחל פולינום (באמצעות __init__) באותה הדרך כמו קודם - ע"י הגדרת רשימת מקדמים. בנוסף, התוצאה של הפעלת __repr__ תהיה זהה לזו שהייתה קודם. במילים אחרות, מבחינת המשתמש אין הבדל "נראה לעין" בעבודה עם פולינומים, למעט העובדה ששם המחלקה השתנה.
למשל:

```
>>> q = Polynomial_sparse([0, 0, 0, 6])
>>> q
(6*x^3)
```

<u>הכוונה</u>: כאשר המשתמש מאתחל פולינום דליל כמו בדוגמת ההרצה שלעיל, כמובן שהאתחול לא יכול לעבוד בפחות מזמן $O(n)$ עבור רשימת מקדמים באורך n. אבל __init__ של Polynomial_sparse תוכל לקבל את הקלט שלה בעוד צורה מלבד זו, ובמקרה כזה זמן הריצה שלה יהיה $O(k)$. derivative תשתמש באיתחול מסוג זה. נסו אם כן לחשוב איזה ייצוג פנימי (שדות) יתאימו לדרישות אלו. שימו לב שיש כאן יותר מפתרון אפשרי יחיד.

<u>סוף</u>