אוניברסיטת תל אביב - בית הספר למדעי המחשב
מבוא מורחב למדעי המחשב, חורף 2020

תרגיל בית מספר 4 - להגשה עד 24.12.19 (יום ג') 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 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 hw4_012345678.pdf ו- hw4_012345678.py.
- בכל השאלות ניתן להניח כי הקלט לתכנית / לפונקציות הינו תקין, אלא אם צוין 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.

שאלה 1

בהרצאה ראינו את אלגוריתם quicksort אשר משתמש הן ברקורסיה והן באקראיות. האקראיות, כזכור, הייתה בבחירת איבר הציר (pivot) שלפיו נחלק את הרשימה לשלוש רשימות (איברים שקטנים, זהים בגודלם וגדולים מהציר שבחרנו).

דיברנו גם על האפשרות לממש את האלגוריתם באופן דטרמיניסטי, כלומר, ללא שימוש באקראיות. ההצעה שלנו למימוש דטרמיניסטי הייתה פשוטה ביותר, איבר הציר יהיה האיבר הראשון ברשימה. להלן המימוש של פונקציה זו:

```
def det_quicksort(lst):
    """ sort using deterministic pivot selection """
    if len(lst) <= 1:
        return lst
    else:
        pivot = lst[0]      # select first element from list
        smaller = [elem for elem in lst if elem < pivot]
        equal = [elem for elem in lst if elem == pivot]
        greater = [elem for elem in lst if elem > pivot]
        return det_quicksort(smaller) + equal + det_quicksort(greater) #two recursive ca
lls
```

כזכור, זמן הריצה הטוב ביותר של quicksort (עם אקראיות) הוא $O(n \log n)$ כאשר n הוא אורך הרשימה, ואילו זמן הריצה הגרוע ביותר הוא $O(n^2)$. בסעיפים הקרובים נסתכל על הגרסא הדטרמיניסטית של האלגוריתם ונחשב את הסבירות שזמן הריצה יהיה הגרוע ביותר, כתלות באופן בו מסודרים איברי הרשימה.

סעיף א'

בהנתן n (טבעי, חיובי), נסמן ב-$p(n)$ את מספר הדרכים בהן ניתן לסדר את איברי הרשימה $lst = [1,2,\dots,n]$.
לדוגמא, עבור $n = 3$ ישנן שש דרכים לסדר את הרשימה:

$$[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]$$

ולכן $p(3) = 6$. מהו $p(n)$ ל-n כללי? הוכיחו את תשובתכם.

סעיף ב'

בהנתן n (טבעי, חיובי), נסמן ב-$w(n)$ את מספר הדרכים בהן ניתן לסדר את איברי הרשימה $lst = [1,2,\dots,n]$ כך שריצת det_quicksort על הרשימה תרוץ בזמן הארוך ביותר.
לדוגמא, עבור $n = 3$ ניתן לוודא כי הסידורים הבאים:

$$[1,2,3],[1,3,2],[3,1,2],[3,2,1]$$

יניבו את זמן הריצה הגרוע ביותר, ולכן $w(3) = 4$. מהו $w(n)$ ל-n כללי? הוכיחו את תשובתכם.

סעיף ג'

בהנתן שני הסעיפים הקודמים, חשבו את הגבול:

$$\lim_{n\to\infty} \frac{w(n)}{p(n)}$$

מה ניתן להסיק מהגבול על הסיכוי לכך שזמן הריצה של האלגוריתם יהיה גרוע ביותר ככל ש-n גדל?

שאלה 2

במשחק "נים" משתתפים שני שחקנים: שחקן א' ושחקן ב'. בתחילת המשחק, ניצבות לפניהם $\boldsymbol{n}$ ערימות נפרדות של גפרורים, כאשר בכל ערימה לפחות גפרור אחד. הניחו כי $\boldsymbol{n > 0}$. הערימות ממוספרות מ-0 עד $\boldsymbol{n-1}$. נסמן את כמות הגפרורים בערימה $\boldsymbol{i}$ בתור $\boldsymbol{a_i}$.
כל שחקן בתורו (שחקן א' מתחיל את המשחק) בוחר ערימה לא ריקה של גפרורים מבין $\boldsymbol{n}$ הערימות, ומוציא ממנה מספר מסוים של גפרורים לבחירתו: לפחות גפרור אחד, אך לא יותר ממספר הגפרורים שבערימה.
המשחק נגמר כאשר לא נותרו יותר גפרורים באף ערימה (כלומר, $\boldsymbol{a_i = 0}$ לכל $\boldsymbol{i}$). המנצח הוא האחרון שהוציא גפרורים מערימה כלשהי.

דוגמא למהלך משחק:
עבור $\boldsymbol{n = 3}$, מספר הגפרורים בערימות הוא $\boldsymbol{a_0 = 1, a_1 = 1, a_2 = 3}$

1. שחקן א' פותח את המשחק, ומוציא מערימה 2 את כל שלושת הגפרורים. כעת $\boldsymbol{a_0 = 1, a_1 = 1, a_2 = 0}$.
2. שחקן ב' מוציא מערימה 1 גפרור אחד. כעת $\boldsymbol{a_0 = 1, a_1 = 0, a_2 = 0}$
3. שחקן א' מוציא מערימה 0 גפרור אחד. כעת לא נותרו גפרורים, ולכן שחקן א' ניצח.

נגדיר את מצב הערימות $\boldsymbol{a}$ כמצב מנצח אם השחקן שתורו לשחק יכול להבטיח לעצמו נצחון (כלומר לא משנה מה יעשה היריב, השחקן שתורו לשחק יכול, אם יפעל בתבונה, לנצח במשחק).
נרצה לבדוק האם מצב ערימות מסוים הינו מצב מנצח באמצעות אלגוריתם רקורסיבי, אשר רץ על כל מצבי המשחק האפשריים. כלומר, בכל צעד רקורסיה, נבחר ערימת גפרורים מסוימת ונוציא ממנה כמות חוקית של גפרורים.

סעיף א'

השלימו בקובץ השלד את הפונקציה can_win_nim אשר מקבלת רשימה lst בגודל חיובי n ובה מספרים שלמים חיוביים בלבד המייצגים את מספר הגפרורים ב-n הערמות. הפונקציה תחזיר $True$ אם"ם מצב הערימות lst הינו מצב מנצח.

דוגמאות הרצה:

```
>>> can_win_nim([3,1,1])
True
>>> can_win_nim([2, 2])
False
>>> can_win_nim([7, 2])
True
```

הנחיות: על הפתרון להיות רקורסיבי.

סעיף ב'

הוסיפו לקובץ ה-pdf צייור של עץ הרקורסיה לקוד שכתבתם עבור $n = 2$,
כאשר מספר הגפרורים בערימות הוא $a_0 = 2, a_1 = 1$. עבור כל צומת בעץ **מלבד** העלים, רשמו $True$ ליד הצומת אם זהו מצב ומנצח ו-$False$ אחרת.

הנחיות: אין לצייר את עץ הרקורסיה ביד. ניתן להשתמש בכלים זמינים ברשת דוגמת draw.io או בתוכנות דוגמת Visio או Open Office Draw.

סעיף ג'

השלימו בקובץ השלד את הפונקציה can_win_nim_mem. בדומה לסעיף א', גם פונקציה זו תקבל רשימה lst בגודל חיובי n ובה מספרים שלמים חיוביים בלבד המייצגים את מספר הגפרורים ב-n הערמות ותחזיר $True$ אם"ם מצב הערימות lst הינו מצב מנצח. בשונה מסעיף א', על הפונקציה כעת להשתמש בממואיזציה.

דוגמאות הרצה:

```
>>> can_win_nim_mem([3, 1, 1])
```

```
True
>>> can_win_nim_mem([2, 2])
False
>>> can_win_nim_mem([7, 2])
True
```

הנחיות:

- על הפתרון להיות רקורסיבי ולהשתמש בממואיזציה (בפרט, כל קונפיגורציה של המשחק תנתן כקלט לפונקציה פעם אחת לכל היותר).
- יש להשתמש ב-can_win_nim_mem כפונקצית מעטפת אשר תקרא לפונקציה נוספת (והיא תהיה הפונקציה הרקורסיבית)

<u>סעיף ד'</u>

נתבונן כעת ברשימה $lst = [3\ for\ i\ in\ range(n)]$ עבור ערכי n הולכים וגדלים.

ציינו בקובץ ה-pdf מהו ה-n הגדול ביותר עבורו can_win_nim(lst) מסיימת לרוץ תוך פחות מדקה ומהו ה-n הגדול ביותר עבורו can_win_nim_mem(lst) מסיימת לרוץ תוך פחות מדקה.

שאלה 3

בשאלה זו נרצה לעבוד עם מטריצות. מאחר שאין לנו בפייתון משתנה מטיפוס מטריצה, נשתמש ברשימות מקוננות על מנת לייצג מטריצות. הייצוג שלנו יהיה טבעי ביותר: בהנתן מטריצה M בגודל $k \times n$ (כלומר, מטריצה בת n שורות ו-k עמודות), נבנה רשימה מקוננת lst_m המכילה n תתי-רשימות, כל אחת באורך k.
כעת, את האיבר $M_{i,j}$ נמקם ב-$lst\_m[i][j]$. להלן דוגמא למטריצה בגודל 2×3:

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

והרשימה שתייצג את M היא הרשימה הבאה:

```
lst_m = [[1, 2, 3], [4, 5, 6]]
```

נגדיר עתה מטריצה **בינארית** מעניינת במיוחד – מטריצת Hadamard. למספר טבעי n מטריצת Hadamard מסדר n (אותה נסמן מעתה ואילך על ידי had(n)) מוגדרת באופן הרקורסיבי הבא:

- עבור n = 0, had(0) היא מטריצה מגודל 1×1 המכילה את האיבר אפס: $had(0) = (0)$
- עבור n > 0, had(n) היא מטריצה ריבועית מגודל $2^n \times 2^n$ אשר בנויה באופן הרקורסיבי הבא:
  אם had(n-1) היא מטריצת Hadamard מסדר n-1, אזי:

$$had(n) = \begin{pmatrix} had(n-1) & had(n-1) \\ had(n-1) & \overline{had(n-1)} \end{pmatrix}$$

כאשר עבור מטריצה בינארית M נסמן ב-$\overline{M}$ את המטריצה שבה מחליפים ערכי 0 ב-M ב-1 וערכי 1 ב-M ב-0.
כלומר, had(n) מורכבת מ-4 עותקים של had(n-1) (זוהי מטריצת בלוקים עם שני בלוקים עליונים של had(n-1) ושני בלוקים תחתונים של had(n-1)), כאשר בבלוק הימני-תחתון ערכי המטריצה מתהפכים (כלומר, ערכי 0 הופכים ל-1 וערכי 1 ל-0).

למטריצת Hadamard התכונה המרתקת הבאה: כל שתי שורות שונות במטריצה נבדלות בדיוק בחצי מהקואורדינטות שלהן (בדקו זאת!). תכונה זו שימושית במיוחד בתחום של תיקון שגיאות, אותו נפגוש בהמשך הקורס.

להלן שלוש המטריצות had(0), had(1), had(2) מיוצגות כרשימות מקוננות על פי הייצוג שקבענו לעיל:

```
had0 = [[0]]
had1 = [[0, 0], [0, 1]]
had2 = [[0, 0, 0, 0], [0, 1, 0, 1], [0, 0, 1, 1], [0, 1, 1, 0]]
```

נעיר כעת כי קשה להבחין במבנה הרקורסיבי באופן בו הצגנו את המטריצות. דרך נוחה לבחון את המטריצות תהיה על ידי הדפסה של כל שורה ברשימה המקוננת בנפרד. בקובץ השלד נמצאת הפונקציה print_mat אשר מקבלת רשימה מקוננת כקלט ומדפיסה את תוכן הרשימה שורה-אחר-שורה. להלן שלושת מטריצות Hadamard הראשונות בהדפסה זו:

```
>>> print_mat(had(0))
[0]
>>> print_mat(had(1))
[0, 0]
[0, 1]
>>> print_mat(had(2))
[0, 0, 0, 0]
[0, 1, 0, 1]
[0, 0, 1, 1]
[0, 1, 1, 0]
```

בתרגיל הנוכחי נממש את הפונקציה had(n) אשר מקבלת כקלט מספר טבעי n ומחזירה את מטריצת Hadamard מסדר n.

<u>סעיף א'</u>

ממשו בקובץ השלד את הפונקציה concat_hor אשר מקבלת כקלט שתי רשימות מקוננות mat1, mat2 ומחזירה **רשימה מקוננת חדשה** המורכבת משרשור אופקי של תתי הרשימות, לדוגמא:

```
>>> mat1 = [[1, 2], [3, 4]]
>>> mat2 = [[5, 6, 7], [8, 9, 10]]
>>> print_mat(concat_hor(mat1, mat2))
[1, 2, 5, 6, 7]
[3, 4, 8, 9, 10]
```

הנחיות:

- על מנת שהשרשור יהיה מוגדר היטב ניתן להניח כי len(mat1) == len(mat2)
- ניתן להניח כי כל תת-רשימה ב-mat1 היא באורך זהה (וכך גם ב-mat2)
- לא ניתן להניח כי תתי הרשימות ב-mat1 הן באורך זהה לתתי הרשימות ב-mat2 (כפי שרואים בדוגמא)
- על הפונקציה לא לשנות את ערכי mat1, mat2

<u>סעיף ב'</u>

ממשו באופן דומה בקובץ השלד את הפונקציה concat_vert שמקבלת כקלט שתי רשימות מקוננות mat1, mat2 ומחזירה **רשימה מקוננת חדשה** המורכבת משרשור אנכי של תתי הרשימות, לדוגמא:

```
>>> mat1 = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]
>>> mat2 = [[11,22,33,44],[55,66,77,88]]
>>> print_mat(concat_vert(mat1, mat2))
```

```
[1, 2, 3, 4]
[5, 6, 7, 8]
[9, 10, 11, 12]
[11, 22, 33, 44]
[55, 66, 77, 88]
```

הנחיות:

- על מנת שהשרשור יהיה מוגדר היטב ניתן להניח כי אורך כל תתי הרשימות הן ב-mat1 והן ב-mat2 זהה (כלומר, תתי הרשימות בשתי הרשימות הן מאורך k בדיוק ל-k כלשהוא)
- לא ניתן להניח כי len(mat1) == len(mat2) (כפי שרואים בדוגמא)
- על הפונקציה לא לשנות את ערכי mat1, mat2

סעיף ג'

ממשו את הפונקציה inv אשר מקבלת כקלט רשימה מקוננת בינארית (כלומר, רשימה מקוננת שכל כניסותיה הן 0 או 1) ומחזירה רשימה מקוננת **חדשה** ההופכית ל-mat (כלומר, מחליפה 0 ב-1 ו-1 ב-0)

דוגמאות הרצה:

```
>>> mat = [[1, 0, 1], [0, 0, 0], [1, 1, 1]]
>>> print(inv(mat))
[[0, 1, 0], [1, 1, 1], [0, 0, 0]]
>>> print_mat(inv(mat))
[0, 1, 0]
[1, 1, 1]
[0, 0, 0]
```

הנחיות:

- על הפונקציה לא לשנות את ערכי mat

סעיף ד'

השתמשו בפונקצית שבנינו בסעיפים א'-ג' על מנת לממש את הפונקציה had(n) אשר מחזירה את מטריצת Hadamard מסדר n.

דוגמאות הרצה:

```
>>> print(had(0))
[[0]]
>>> print(had(1))
[[0, 0], [0, 1]]
>>> print(had(2))
```

[[0, 0, 0, 0], [0, 1, 0, 1], [0, 0, 1, 1], [0, 1, 1, 0]]

הנחיות:

- על המימוש להיות רקורסיבי
- על המימוש להשתמש בשלוש הפונקציות שבנינו בסעיפים הקודמים

שאלה 4

בתרגול דיברנו על בעיית Subset sum. להלן תזכורת – בהנתן רשימה של מספרים L ומספר s, נרצה לקבוע האם קיימת תת רשימה $R \subseteq L$ כך שסכום איברי R הוא s: $\sum_{x \in R} x = s$.

להלן הפונקציה הרקורסיבית שכתבנו על מנת לפתור את הבעיה:

```
def subset_sum(L, s):
    if s == 0:
        return True
    if L == []:
        return False

    with_first = subset_sum(L[1:], s - L[0])
    without_first = subset_sum(L[1:], s)

    return with_first or without_first
```

סעיף א'

1. ציירו בקובץ ה-PDF את עץ הרקורסיה של subset_sum([1, 2, 4, 8], 13).
   הנחיה – אין לכתוב את עץ הרקורסיה בכתב יד. ניתן להשתמש בכלים זמינים ברשת דוגמת draw.io או בתוכנות דוגמת Visio או Open Office Draw.
2. בהנתן n טבעי תנו דוגמא לרשימה L **<u>ללא חזרות</u>** באורך n ולמספר s (s יכול להיות תלוי או בלתי תלוי ב-n, כרצונכם) כך שיש t תתי רשימות שונות ב-L שסכומן הוא s וכמו כן $n = O(t)$.

סעיף ב'

הפונקציה subset_sum היא דוגמא **<u>לפונקצית הכרעה</u>**. בהינתן בעיה מסוימת (במקרה שלנו – האם קיימת תת רשימה R ב-L שסכומה s), הפונקציה מחזירה True אם קיים פיתרון לבעיה ו-False אחרת.

לעתים אנחנו כמובן לא רק נתעניין בשאלה "האם קיים פתרון?" אלא גם בחיפוש של פתרון קונקרטי לבעיה.

במקרה של Subset sum, פתרון קונקרטי לבעיה הוא תת רשימה R ב-L שסכומה הוא s או None במקרה שבו אין תת רשימה שכזו. לפונקציה שמחזירה פתרון קונקרטי לבעיה נקרא **<u>פונקצית חיפוש</u>**.

השלימו את פונקצית החיפוש subset_sum_search(L, s) שבקובץ השלד על פי ההוראות שתיארנו זה עתה:

- אם קיימת תת רשימה R ב-L שסכומה s הפונקציה תחזיר את R
- אחרת, הפונקציה תחזיר None

הנחיות וטיפים:

- המימוש של הפתרון חייב להיות רקורסיבי
- ניתן (ורצוי) להשתמש בפונקציה subset_sum_search כפונקצית מעטפה שתקרא לפונקציה הרקורסיבית שאותה תכתבו בעצמכם
- אם יש יותר מפתרון אחד לבעיה, כלומר, לפחות שתי תתי רשימות שונות R ו-S ב-L שסכומן s, ניתן להחזיר כל אחת מתתי הרשימות כפתרון

- ניתן להניח כי כל המספרים ב-L שונים
- אין צורך להחזיר את איברי תת הרשימה על פי הסדר המקורי שלהם. למשל, עבור $L = [1, 2, 3]$ ו-$s = 5$ ניתן להחזיר את הרשימה $R = [3, 2]$ וכן את הרשימה $R = [2, 3]$.

דוגמאות הרצה:

```
>>> subset_sum_search([1, 3, 8], 11)
[3, 8]
>>> subset_sum_search([], 0)
[]
>>> subset_sum_search([1, 2], 0)
[]
>>> subset_sum_search([1, 3, 8, 6, 5], 11)
[3, 8] # also [8, 3] or [6, 5] or [5, 6] are valid solutions
```

שאלה 5

בשאלה זו 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 in 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
```

שאלה 6

בשאלה זו נממש אלגוריתמים להשוואה בין מחרוזות.

במהלך השאלה, אין להשתמש באופרטור ההשוואה בין מחרוזות אבל מותר להשוות תווים בודדים בין מחרוזות.

כדוגמה, אם $s1, s2$ הן מחרוזות ו-i הוא שלם, ההשוואות הבאות מותרות:

```
s1[i]==s2[i+1], s1[0]==s2[-1]
```

וההשוואות הבאות אסורות:

```
s1==s2, s1[2:4]==s2[1:3], s1==""
```

בנוסף, לאורך השאלה אין להשתמש במתודות מובנות של המחלקה str (למשל: $contains$) אך מותר לבצע סלייסינג על מחרוזות ולבדוק מהו האורך שלהן.

ניתן להניח כי המחרוזות בשאלה מכילות רק אותיות קטנות (lowercase) מהא״ב האנגלי (a-z).

סעיף א׳

ממשו בקובץ השלד את הפונקציה comp, אשר מקבלת שתי מחרוזות $s1, s2$ ומחזירה $True$ אם״ם המחרוזות זהות.

הנחיות: על הפתרון להיות רקורסיבי

דוגמאות הרצה:

```
>>> comp('ab','ab')
True
>>> comp('','')
True
>>> comp('a','ab')
False
```

סעיף ב׳

כעת נממש גרסה מורחבת ל-$comp$, בשם $comp\_ext$, שבה נרשה למחרוזת s1 להכיל שני תווים מיוחדים נוספים (מלבד אותיות הא״ב האנגלי a-z).

1. s1 יכולה להכיל את התו '+' (0 או יותר פעמים), והוא מפורש כתו מיוחד, שמתאים לכל תו יחיד ב s2.

דוגמאות הרצה של הפונקציה המורחבת:

```
>>> comp_ext('+','a')
True
>>> comp_ext('+','')
False
>>> comp_ext('a+b','axb')
True
>>> comp_ext('a+b','axxb')
False
>>> comp_ext("ab+d+f", "abcdef")
True
>>> comp_ext("ab+d+f", "abcdfe")
False
```

2. s1 יכולה להכיל את התו '*' (0 או יותר פעמים), והוא מפורש כתו מיוחד שמתאים לכל רצף של תווים באורך 0 או יותר ב s2.

דוגמאות הרצה נוספות של הפונקציה המורחבת:

```
>>> comp_ext("abc*xyz", "abcxyz")
True
>>> comp_ext("abc*xyz", "abcdddddxyz")
True
>>> comp_ext("abc*xyz", "abcxyzz")
False
>>> comp_ext("a***", "a")
True
```

שימו לב כי $s1$ יכולה להכיל את שני התווים המיוחדים יחדיו:

```
>>> comp_ext('abc+d*e', 'abcxdxxxe')
True
>>> comp_ext('abc+d*e', 'abcxde')
True
>>> comp_ext('abc+d*e', 'abcxdef')
False
```

הנחיות: על הפתרון להיות רקורסיבי.

<u>סוף</u>