תרגיל בית 7 :

<u>הנחיות כלליות:</u>

- קראו בעיון את השאלות והקפידו שהתוכניות שלכם פועלות בהתאם לנדרש.
- את התרגיל יש לפתור לבד!
- הקפידו על כללי ההגשה המפורסמים באתר. בפרט, יש להגיש את כל השאלות יחד בקובץ ex7_012345678.py המצורף לתרגיל, לאחר החלפת הספרות 012345678 במספר ת.ז. שלכם, כל 9 הספרות כולל, ספרת ביקורת.
- <u>אופן ביצוע התרגיל:</u> בתרגיל זה עליכם להשלים את הקוד בקובץ המצורף.
- <u>בדיקה עצמית:</u> כדי לוודא את נכונותן ואת עמידותן של התוכניות לקלטים, בכל שאלה הריצו את תוכניתכם עם מגוון קלטים שונים, אלה שהופיעו כדוגמאות בתרגיל וקלטים נוספים עליהם חשבתם (וודאו כי הפלט נכון).
- השאלות נבדקות באופן אוטומטי. לכן, עליכם לרשום את הקוד שלכם אך ורק במקומות המתאימים לכך בקובץ השלד.
- ניתן להניח כי הקלט שמקבלות הפונקציות תקין (אלא אם נכתב אחרת).
- אין לשנות שמות פונקציות או משתנים שקיימים בקובץ השלד של התרגיל.
- אין למחוק את ההערות שמופיעות בשלד.
- אין להשתמש בספריות חיצוניות (אסור לעשות **import**), אלא אם כתוב במפורש אחרת.
- נושא התרגיל הוא רקורסיה וממואיזציה. **<u>פתרונות לא רקורסיביים לא יתקבלו.</u>** שימו לב שבכל הפונקציות, עליכם להחזיר ערכים ולא להדפיס.
- <u>מועד אחרון להגשה:</u> כמפורסם באתר.

הערה כללית: תרגיל זה עוסק במימושים רקורסיבים. שימו לב שבכל מחשב וברוב התוכנות איתם תעבדו (למשל IDLE, PyCharm) ישנה מגבלה לכמות הקריאות הרקוריסיביות שניתן לבצע עבור אותה פונקציה ברצף על מנת למנוע רקורסיות אין סופיות. לכן, יתכן כי עבור ריצות שידרשו מספר גבוה מאוד של קריאות רקורסיות תקבלו שגיאה מהסוג הבא :

```
>>> def a(x):
        a(x)


>>> a(5)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    a(5)
  File "<pyshell#3>", line 2, in a
    a(x)
  File "<pyshell#3>", line 2, in a
    a(x)
  File "<pyshell#3>", line 2, in a
    a(x)
  [Previous line repeated 990 more times]
RecursionError: maximum recursion depth exceeded
```

למרות שהקוד שלכם תקין. בתרגיל זה אתם יכולים להתעלם ממצבים אלה. לא נבדוק את הקוד שלכם עם קלטים שידרשו קריאה למספר גדול מאוד של קריאות רקורסיביות שעלול ליצור שגיאה כזו במימוש סביר ו\או הגיוני של הבעיה. היה ונתקלתם בשגיאה כזו, ניתן לשנות את עומק הרקורסיה המקסימלי בפייתון מתוך ה-**shell** על ידי הרצת הפקודות הבאות:

```
>>> import sys

>>> sys.setrecursionlimit(10000)
```

שאלה 1

בשיעור ראינו את הסדרה פיבונאצ'י בה האיבר באינדקס 0 שווה ל-0 ,האיבר באינדקס 1 שווה ל-1 וכל איבר בהמשך שווה לסכום שני קודמיו. האיברים הראשונים בסדרת פיבונאצ'י נראים כך:

0,1,1,2,3,5,8,13,21,34…

באופן דומה, ניתן להגדיר את סדרת "פור-בונאצ'י" בה האיבר באינדקס 0 שווה ל-0 ,האיבר באינדקס 1 שווה ל-1 ,האיבר באינדקס 2 שווה ל-2 והאיבר באינדקס 3 שווה ל3.

כל איבר בהמשך שווה לסכום ארבעת קודמיו. האיברים הראשונים בסדרת "פור-בונאצ'י" נראים כך:

0,1,2,**3**,6,12,23,44,85…

האיבר באינדקס 4 שווה ל- 3+2+1+0=6, האיבר באינדקס 5 שווה ל- 6+3+2+1=12, וכן הלאה.

סעיף א'

ממשו את הפונקציה four_bonacci_rec(n) , אשר מקבלת את המספר השלם n, ומחזירה את האיבר באינדקס n בסדרת "פור-בונאצ'י".

- המימוש צריך להיות **רקורסיבי וללא שימוש בממואיזציה**.
- ניתן להניח ש- n>=0 ושלם.

סעיף ב'

ממשו את הפונקציה four_bonacci_mem(n,mem=None) , אשר מקבלת את המספר השלם n, ומחזירה את האיבר באינדקס n בסדרת "פור-בונאצ'י". פונקציה זו משתמשת ב**ממואיזציה** בכדי למנוע חישובים חוזרים, ובכך משפרת את זמני הריצה של הפונקציה. לכן הפונקציה מקבלת ארגומנט נוסף – memo.

- המימוש צריך להיות **רקורסיבי ועם שימוש בממואיזציה**.
- ניתן להניח ש- n>=0 ושלם.

דוגמאות הרצה:

```
four_bonacci_rec(0)
```

```
0
```

```
four_bonacci_mem(0)
```

```
0
```

```
four_bonacci_rec(1)
```

```
1
```

```
four_bonacci_mem(2)
```

```
2
```

```
four_bonacci_rec(3)
```

```
3
```

```
four_bonacci_mem(5)
```

```
12
```

```
four_bonacci_rec(6)
```

```
23
```

```
four_bonacci_mem(100) #Can be done in a reasonable time only with memoization
```

```
1413851827268925536570438396 0
```

העשרה (אין צורך להגיש):

השוו בין זמני הריצה של שתי הפונקציות עבור n שונים (בדוגמא נבחר n=28), בעזרת קטע הקוד הבא:

```
from timeit import default_timer as timer
n = 28
start = timer ( )
four_bonacci_rec(n=n)
end =  timer ( )
print('Time without memoization for ',n,':',end - start)
start = timer ( )
four_bonacci_mem(n=n)
```

```
end = timer ()
print('Time with memoization for ',n,':',end - start)
```

שאלה 2

הזכרו בשאלה 4 מהתרגיל הקודם: עליכם לטפס במעלה גרם מדרגות בן n מדרגות (n>0 ושלם). בכל צעד טיפוס אתם יכולים לבחור אם לעלות מדרגה אחת בלבד או שתי מדרגות בבת אחת. בכמה דרכים שונות ניתן לטפס לקצה גרם המדרגות?

כתבו את הפונקציה ה**רקורסיבית** climb_combinations_memo(n, memo=None) שמקבלת את n ומחזירה את מספר האפשרויות לעלות את גרם המדרגות. פונקציה זו משתמשת ב**ממואיזציה** בכדי למנוע חישובים חוזרים, ובכך משפרת את זמני הריצה של הפונקציה שכתבתם בתרגיל הקודם. לכן הפונקציה מקבלת ארגומנט נוסף – memo.

דוגמאות הרצה:

```
climb_combinations_memo(1)
```

```
1
```

```
climb_combinations_memo(2)
```

```
2
```

```
climb_combinations_memo(7)
```

```
21
```

```
climb_combinations_memo(42)
```

```
433494437
```

שאלה 3

מספרי *קטלן* הם סדרה של מספרים טבעיים המופיעה בבעיות שונות בקומבינטוריקה.

לפניכם הנוסחה הרקורסיבית לחישוב איבר כללי בסדרת קטלן:

$$C_0 = 1$$

$$C_{n+1} = \sum_{i=0}^{n} C_i \, C_{n-i}$$

אלו שמונת מספרי קטלן הראשונים (שימו לב שהספירה מתחילה מ-0, ולכן האיבר האחרון הוא C_7):

1, 1, 2, 5, 14, 42, 132, 429

ממשו את הפונקציה ה**רקורסיבית** catalan_rec(n,mem=None) המקבלת מספר שלם n>=0, ומחזירה את מספר קטלן ה-n. פונקציה זו משתמשת ב**ממואיזציה** בכדי למנוע חישובים חוזרים, ובכך משפרת את זמני הריצה של הפונקציה. לכן הפונקציה מקבלת ארגומנט נוסף – memo.

- פונקציה זו תשתמש **בנוסחה הרקורסיבית המצורפת למעלה**, ולא בנוסחאות אחרות לחישוב מספרי קטלן.
- הדרכה: שימו לב שבחישוב מספרי קטלן אנו משתמשים מספר פעמים בחישוב מספרי קטלן הקודמים. ניתן לשמור ערכים אלו במילון, כפי שראיתם בשיעור ובתרגול.
- ניתן להניח ש- n>=0 ושלם.
- העשרה (רשות): ניתן לקרוא עוד על מספרי קטלן.

דוגמאות הרצה:

```
catalan_rec(0)
```

```
1
```

```
catalan_rec(1)
```

```
1
```

```
catalan_rec(2)
```

```
2
```

```
catalan_rec(3)
```

```
5
```

```
catalan_rec(4)
```

```
14
```

```
catalan_rec(42)
```

```
3904442991190444395924 0
```

שאלה 4

בעליה של המכולת השכונתית מתעניין בשאלה בכמה דרכים אפשר לפרוט סכום כסף n בעזרת רשימת מטבעות lst.

לדוגמא, את n=5 ניתן לפרוט ב-4 דרכים עם המטבעות lst=[1,2,5,6] - מטבע אחד של 5, חמישה מטבעות של 1, שני מטבעות של 2 ומטבע של 1, שלושה מטבעות של 1 ומטבע של 2, דהיינו: [1,1,1,2], [1,2,2], [1,1,1,1,1], [5]. אם היינו רוצים לפרוט את n=4 עם אותם מטבעות, היו 3 דרכים אפשריות: [1,1,2], [2,2], [1,1,1,1].

<u>הבהרות:</u>

- שימו לב שהספירה צריכה להתבצע כך שאין חשיבות לסדר המטבעות בפריטה, אלא רק לכמה פעמים מופיע כל מטבע בפריטה. למשל: [1,2,2] שקול ל-[2,1,2] ול-[2,2,1], ולכן ויספרו פעם אחת בלבד.
- הניחו שלבעל המכולת יש מלאי בלתי מוגבל של כל אחד מהמטבעות ברשימה lst.
- הניחו שהרשימה lst כוללת רק מספרים גדולים ממש מאפס ושלמים, וכן כל איבר ברשימה הוא ייחודי (דהיינו, לא חוזר על עצמו).

- הניחו כי n הוא שלם.
- במידה ו-lst ריק, אם n = 0 יש להחזיר 1, אחרת 0 (חשבו מדוע).

סעיף א'

ממשו את הפונקציה find_num_changes_rec(n,lst) , אשר מקבלת את המספר השלם n, ורשימת מטבעות lst, ומחזירה את מספר הדרכים שניתן לפרוט את n בעזרת המטבעות ברשימה lst.

- המימוש צריך להיות **רקורסיבי וללא שימוש בממואיזציה**.
- רמז: בדומה לתרגילים שראינו בכיתה, בכל שלב, נתבונן באיבר האחרון ברשימת המטבעות, ונבחר האם לכלול אותו בפריטה או לא (חשבו היטב על צעד הרקורסיה). לדוגמא, אם רשימת המטבעות כוללת את המטבעות הבאים: [1,2,5], נבחר אם להשתמש במטבע 5, או שנחליט לוותר עליו. במידה ונוותר עליו, לא נשתמש במטבע זה בצעדים הבאים. שימו לב שניתן להשתמש בכל מטבע יותר מפעם אחת, ולכן עליכם לוודא שהפתרון הרקורסיבי שלכם מאפשר זאת. כמו כן, חישבו מה קורה במקרים הבאים: כאשר n=0 (יחזיר 1, כי נניח שלא לעשות כלום זו דרך פריטה אחת), כאשר lst=[] וכאשר n<0.
- שימו לב שכאשר אין דרך לפרוט את סכום הכסף n, בעזרת המטבעות ב-lst, יש להחזיר 0 (התבוננו היטב בדוגמאות ההרצה).

סעיף ב'

ממשו את הפונקציה find_num_changes_mem(n,lst) , אשר מקבלת את המספר השלם n, ורשימת מטבעות lst, ומחזירה את מספר הדרכים שניתן לפרוט את n בעזרת המטבעות ברשימה lst. פונקציה זו משתמשת ב**ממואיזציה** בכדי למנוע חישובים חוזרים, ובכך משפרת את זמני הריצה של הפונקציה. לכן הפונקציה מקבלת ארגומנט נוסף – memo.

- המימוש צריך להיות **רקורסיבי ועם שימוש בממואיזציה**.
- רמז: שימו לב שבשמירה לתוך המילון תצטרכו להשתמש ב-tuple כ-key (חישבו למה).

דוגמאות הרצה:

```
find_num_changes_rec(5,[5,6,1,2])
```

```
4
```

```
find_num_changes_rec(-1,[1,2,5,6])
```

```
0
```

```
find_num_changes_rec(1,[2,5,6])
```

```
0
```

```
find_num_changes_rec(4,[1,2,5,6])
```

```
3
```

```
find_num_changes_mem(5,[1,2,5,6])
```

```
4
```

```
find_num_changes_mem(-1,[1,2,5,6])
```

```
0
```

```
find_num_changes_mem(5,[1,2,5,6])
```

```
4
```

```
find_num_changes_mem(1,[2,5,6])
```

```
0
```

```
find_num_changes_mem(4,[1,2,5,6])
```

```
3
```

```
find_num_changes_mem(1430,[1,2,5,6,13]) # Cannot be done in a reasonable time without memoization
```

```
231919276
```

העשרה (אין צורך להגיש):

השוו בין זמני הריצה של שתי הפונקציות עבור n ו-lst שונים (בדוגמא נבחר lst=[1,2,5,6,13],n=143), בעזרת קטע הקוד הבא:

```
from timeit import default_timer as timer
lst = [1,2,5,6,13]
n = 143
start = timer()
find_num_changes_rec(n,lst)
end =  timer()
print('Time without memoization for ',n,lst,':',end - start)
start = timer()
find_num_changes_mem(n,lst)
end = timer()
print('Time with memoization for ',n,lst,':',end - start)
```