תרגיל בית 6+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 ex67_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 לכתוב את הפונקציות במקומות המתאימים בקובץ השלד (החליפו את המילה pass במימוש שלכם).
- <u>מועד אחרון להגשה:</u> כמפורסם באתר.
- <u>בדיקה עצמית:</u> כדי לוודא את נכונותן ואת עמידותן של התוכניות בכל שאלה הריצו את תוכניתכם עם מגוון קלטים שונים וודאו כי הפלט נכון. לנוחיותכם מצורפות בדיקות בסוף הקובץ השלד (אך ייתכנו עוד מקרים שיש לבדוק). **יש לבצע את כל הבדיקות בתוך ה-if:**

  if __name__ == '__main__':
- ניתן להניח כי הקלט שמקבלות הפונקציות הינו כפי שהוגדר בכל שאלה ואין צורך להתייחס לקלט לא תקין, אלא אם כן נאמר אחרת.
- אין למחוק את ההערות שמופיעות בשלד.
- אין לשנות את החתימות (שורות ההגדרה) של הפונקציות שמופיעות בקובץ השלד של התרגיל. עם זאת, אתם רשאים להוסיף פונקציות נוספות ולקרוא להן.
- <u>אין להשתמש בקריאה לספריות חיצוניות (אסור לעשות **import**), אלא אם כן נאמר אחרת.</u>

הנחיות מיוחדות לתרגיל זה:

- **על כל הפונקציות שתכתבו להיות רקורסיביות - פתרונות לא רקורסיביים לא יתקבלו.**
- **קובץ שאלות זה כולל שני תרגילים בתוכו, ליד כל שאלה ייכתב באופן מפורש לאיזה תרגיל הוא שייך, כך שגם אם לא תגישו כלל סעיפים הקשורים לתרגיל 7, עדיין תוכלו לקבל את מלוא הניקוד על תרגיל 6, במידה ופתרתם אותו נכון. החלוקה לתרגילים תהיה לפי סעיפים וליד כל סעיף יצויין באופן מפורש לאיזה תרגיל הוא משוייך.**
- **בסעיפים המשויכים לתרגיל 6 אסור להשתמש בלולאות כלל (מלבד שאלה 3.א). בתרגיל 7, לעומת זאת, ניתן להשתמש בלולאות (מלבד שאלה 5.ב.).**

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…

בשאלה זו נגדיר את הסדרות **"ת'ריבונאצ'י" (threebonacci) ו"פייבונאצ'י" (five_bonacci).**

הסדרה "ת'ריבונאצ'י" (threebonacci) תוגדר באופן הבא: האיבר באינדקס 0 שווה ל-0 ,האיבר באינדקס 1 שווה ל-1 ,האיבר באינדקס 2 שווה ל-2.

כל איבר בהמשך שווה לסכום שלושת קודמיו. האיברים הראשונים בסדרת "ת'ריבונאצ'י" נראים כך:

0,1,2,3,6,11,20,37,68…

האיבר באינדקס 4 שווה ל- 3+2+1=6, האיבר באינדקס 5 שווה ל- 6+3+2=11, וכן הלאה.

הסדרה " פייבונאצ'י" (five_bonacci) תוגדר באופן הבא: את הסדרה "פייבונאצ'י" בה האיבר באינדקס 0 שווה ל-0 ,האיבר באינדקס 1 שווה ל-1 ,האיבר באינדקס 2 שווה ל-2, האיבר באינדקס 3 שווה ל-3 והאיבר באינדקס 4 שווה ל-4.

כל איבר בהמשך שווה לסכום חמשת קודמיו. האיברים הראשונים בסדרת "פייבונאצ'י" נראים כך:

0,1,2,3,4,10,20,39,76,149…

סעיף א' *(תרגיל 6)*

ממשו את הפונקציה threebonacci_rec(n) , אשר מקבלת את המספר השלם n, ומחזירה את האיבר באינדקס n בסדרת "ת'ריבונאצ'י".

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

סעיף ב' (תרגיל 7)

ממשו את הפונקציה five_bonacci_mem(n,memo=None) , אשר מקבלת את המספר השלם n, ומחזירה את האיבר באינדקס n בסדרת "פייבונאצ'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:

```
threebonacci_rec(0)
```

```
0
```

```
threebonacci_rec(12)
```

```
778
```

```
threebonacci_rec(25)
```

```
2145013
```

```
five_bonacci_mem(0)
```

```
0
```

```
five_bonacci_mem(6)
```

```
20
```

```
five_bonacci_mem(100) # Won't be done in reasonable time without memoization
```

```
7756797428851438607520266192 4
```

העשרה (אין צורך להגיש):

ממשו את הפונקציה five_bonacci_rec_no_mem(n) כמו בסעיף ב', רק ללא ממואיזציה.

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 ()
five_bonacci_rec_no_mem(n=n)
end =  timer ()
print('Time without memoization for ',n,':',end - start)
start = timer ()
five_bonacci_mem(n=n)
end = timer ()
print('Time with memoization for ',n,':',end - start)
```

שאלה 2

סעיף א' *(תרגיל 6)*

עליכם לטפס במעלה גרם מדרגות בן n (מספר שלם גדול מאפס) מדרגות. 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 הרקורסיבית climb_combinations(n) שמקבלת מספר שלם גדול מאפס n ומחזירה את מספר האפשרויות לעלות גרם המדרגות בו יש n מדרגות.

דוגמאות הרצה:

```
>>>climb_combinations(3)
3
```

הסבר: יש בדיוק 3 דרכים לעלות 3 מדרגות:

- לעלות מדרגה אחת 3 פעמים
- לעלות שתי מדרגות בבת אחת ואז לעלות מדרגה אחת
- לעלות מדרגה אחת ואז לעלות שתי מדרגות בבת אחת

סעיף ב' *(תרגיל 7)*

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 בסעיף 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

סעיף א' *(תרגיל 6, בתרגיל זה **ניתן** להשתמש בלולאות)*

ממשו את הפונקציה ה**רקורסיבית** catalan_rec(n) המקבלת מספר שלם n>=0, ומחזירה את מספר קטלן ה-n.

סעיף ב' *(תרגיל 7)*

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.

- פונקציות אלו ישתמשו **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' *(תרגיל 6)*

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' *(תרגיל 7)*

ממשו את הפונקציה find_num_changes_mem(n,lst,memo=None) , 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).

פייתון למהנדסים 0509-1820 , סמסטר ב' תשפ"ב 2021-2022

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
```

הדוגמה הבאה תרוץ הרבה מאוד זמן בלי ממואיזציה:

```
>>>find_num_changes_mem(900, [5,6,7,8,9,10])
36808606
```

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=180), בעזרת קטע הקוד הבא:

```
from timeit import default_timer as timer
lst = [1,2,5,6,13]
n = 180
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)
```

שאלה 5

בתור מתכנתים, אנחנו מתעסקים המון עם מידע, אשר יכול להגיע בצורות שונות ומשונות. במקרים מסוימים, נוח מאד לייצג את המידע בצורה של עץ. צורה כזו יכולה להגיע באופן טבעי מהצורה בה המידע מסודר, לדוגמה עצי תיקיות במחשב, קבצי HTML, ועוד (לרוב בנתונים בהם קיימת היררכיה מסוימת).

במהלך הקורס, למדנו בין היתר על רשימות ומילונים בפייתון, אלו מאפשרים לנו לשמור מידע ולגשת אליו בצורה מסוימת, המוגדרת בשפה. עצים, לעומת זאת, הם מבנה נתונים מופשט, אשר אינו מוגדר בשפה אך נותן לנו לסדר בצורה מסוימת את המידע ולבצע עליו פעולות לפי הסידור הזה.

אנחנו מאד אוהבים הגדרות, אז הרי לכם כמה:

- עץ מורכב מ**צמתים** ו**קשתות** המחברות בין הצמתים.
- אנו מסתכלים על העץ מלמעלה למטה, כאשר הצמת העליון ביותר נקרא ה**שורש**.
- לכל צמת בעץ, ובפרט גם לשורש, יכולים להיות **בנים** – צמתים המחוברים אליו בקשתות מ**למטה**.
- ה**אב** של צמת הוא הצמת אליו הוא מחובר מ**למעלה**. לשורש אין **אב**.
- שני צמתים בעלי אב משותף נקראים **אחים**.
- צמתים ללא בנים נקראים **עלים**.

בדרך כלל, צמתים מציינים משהו – למשל מחזיקים בתוכם ערך מספרי כלשהו. בצורה כזו, אנחנו יכולים לסדר את הנתונים שלנו על גבי העץ, ולהשתמש באלגוריתמים הרבים אשר פותחו בעבור עצים.

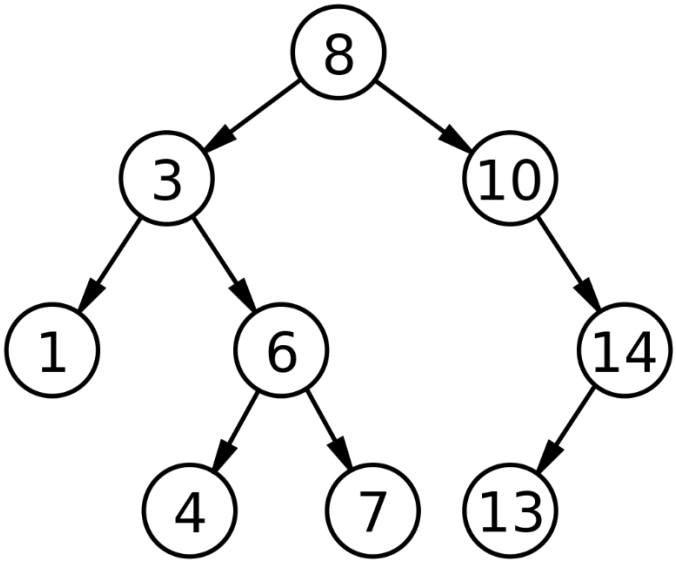
בתרגיל זה נדון אך ורק בעצים **בינאריים**, כלומר עצים בהם לכל צמת יש **לכל היותר** שני בנים. בפרט, נדון **בעצים בינאריים ממוינים** – בעץ שכזה, הערך השמור בכל צמת יהיה **גדול** מהערך ששמור בבנו **השמאלי** **ומהערכים השמורים בכל צאציו של הבן השמאלי**, אך **קטן** מהערך ששמור בבנו **הימני** **ומהערכים השמורים בכל צאציו של הבן הימני**.

מומלץ לקרוא את הערכים הבאים:

https://en.wikipedia.org/wiki/Tree_(data_structure)

https://en.wikipedia.org/wiki/Binary_search_tree

דוגמה לעץ בינארי ממוין (לקוח מויקיפידיה):



ניתן לראות שבשורש שמור המספר 8, בבנו השמאלי של השורש שמור המספר 3, ובימני 10. כלומר החוקיות של העץ אכן מתקיימת על השורש. האם היא מתקיימת עבור כל שאר הצמתים האחרים בעץ? וודאי! לדוגמה 1<3<6, 10<14 וכדומה.

למטרת התרגיל, נגדיר לכל צמת מספר סידורי (מס"ד), לפי הסדר הבא: השורש – 0, בנו השמאלי 1, בנו הימני 2 וכך הלאה, כך שהמס"ד של הבן השמאלי של הצומת עם המס"ד n, יהיה 2n+1, ושל הימני 2n+2. שימו לב שלפי הגדרה זו לכל צמת בעץ קיים בכרח מס"ד ייחודי אך לא כל המס"דים קיימים באופן רציף לכל עץ (חישבו מדוע).

כאמור, עץ הוא מבנה נתונים מופשט, כלומר אין אנחנו יכולים להחזיק בפייתון עץ, אלא אם נמצא דרך לייצג אותו בעזרת טיפוסים הקיימים בשפה. בתרגיל זה נבחר לעשות זאת באמצעות מילון, באופן הבא: מפתח במילון הוא המס"ד של צמת קיים בעץ, והערך באותו מפתח הוא הערך השמור בצמת.

לדוגמה, העץ בתרשים למעלה מיוצג על ידי המילון הבא::

```
>> tree = {0: 8, 1: 3, 2: 10, 3: 1, 4: 6, 9:4, 10: 7, 2: 10, 6:14, 13: 13}
```

סעיף א' *(תרגיל 6)*

אחד השימושים המעניינים של עץ בינארי ממוינים הוא חיפוש ערכים. עליכם לכתוב את הפונקציה **הרקורסיבית** isInTree(tree, val, idx=0) המקבלת את העץ, ערך מספרי לחפש בעץ, ומס"ד, השייך לצמת בעץ עליו אשר ממנו מתבצעת הבדיקה. הפונקציה תחזיר True במקרה שהערך נמצא בעץ, ו-False אחרת. לדוגמה:

```
>> tree = {0: 5, 1: 2}

>> isInTree(tree, 5)

>> True

>> isInTree(tree, 16)

>> False
```

סעיף ב' *(תרגיל 7. בתרגיל זה **אסור** להשתמש בלולאות)*

עליכם לכתוב את הפונקציה **הרקורסיבית** create_sorted_binary_tree(lst, tree=None, idx=0) אשר מקבלת רשימה לא ממוינת של מספרים ומחזירה את העץ הבינארי הממוין הכולל את הערכים שבתוכה.

הנחיה חשובה: עליכם לבנות את העץ הבינארי הממוין על ידי לקיחת האיבר הבא ברשימה **לפי סדר האיברים**, ומבלי לשנות את העץ בין הכנסת איברים. כלומר, עבור הרשימה [5,4,10], האיברים יוכנסו בסדר הבא: קודם 5, לאחר מכן 4 ורק בסוף 10, כך שמיקומי האיברים 5,4 לפני הכנסת האיבר 10 ואחריה – נותרו ללא שינוי.

לדוגמה:

```
>> lst = [10, 3, 8]

>> create_sorted_binary_tree(lst)

>> {0: 10, 1: 3, 4: 8}


>> lst = [12, 23, 66, 15, 54, 1, 84]

>> create_sorted_binary_tree(lst)

>> {0: 12, 2: 23, 6: 66, 5: 15, 13: 54, 1: 1, 14: 84}
```

הערה: ניתן להניח שערכים ברשימה הנתונה לא חוזרים על עצמם.