תאריך המבחן:	02.02.2022
שמות המרצים:	ד"ר אסף זריצקי
שמות המתרגלים:	מר ישעיה צברי, גב' יעל מה-טוב, גב' שיר כהן
שם הקורס:	מבוא למדעי המחשב בשפת פייתון
מספר הקורס:	38211111 ,37211111
שנה:	2022, מועד א'
משך הבחינה:	שלוש שעות
חומר עזר:	דף A4 כתוב בכתב יד בשני הצדדים

אנא קראו היטב את ההוראות שלהלן:

- במבחן **3** שאלות הכוללות סעיפי משנה. במבחן **103** נקודות. כדי לקבל את מלוא הניקוד יש לענות נכון על כל השאלות. ניקוד כל סעיף מצוין לידו. אין בהכרח קשר בין ניקוד הסעיף ובין רמת הקושי שלו.
- מומלץ לקרוא כל שאלה עד סופה, על כל סעיפיה, לפני תחילת הפתרון.
- את הפתרונות יש לכתוב במסגרות המסומנות לכל שאלה בטופס הבחינה. המחברת שקיבלתם היא מחברת טיוטה, והיא לא תימסר כלל לבדיקה. **בסיום הבחינה נאסוף אך ורק את דף התשובות**. כל שאר החומר יועבר לגריסה.
- בכל סעיף ניתן להשתמש בקוד שהתבקשתם לכתוב בסעיפים הקודמים, **גם אם לא פתרתם אותם**.
- **ניתן להניח שהקלט תקין**, אלא אם נכתב אחרת בשאלה.
- במידה ואינכם יודעים את התשובה לסעיף שלם כלשהו, רשמו "לא יודע/ת" (במקום תשובה) ותזכו ב-20% מניקוד הסעיף. אם רשום "לא יודע/ת", ההתייחסות היא לכל הסעיף.
- אין להשתמש בחבילות או במודולים, אלא אם נאמר במפורש. כאשר אתם מתבקשים להשתמש בחבילה אין צורך לבצע import.

בהצלחה!

שאלה 1 (30 נקודות)

בשאלה זאת תממשו בעצמכם מספר פונקציות שקיים להן מימוש מובנה בפייתון.
כמובן שאין להשתמש בפונקציות המובנות שהינכם נדרשים לממש (accumulate, zip, zip_longest).

סעיף א' (5 נקודות)

ממשו את הפונקציה my_accumulate (itr, func, init =0) אשר מקבלת כקלט שלושה ארגומנטים:

- משתנה מטיפוס Iterable בשם itr.
- משתנה מטיפוס פונקציה, בשם func, אשר מקבלת שני ארגומנטים ומחזירה ערך יחיד.
- משתנה מטיפוס int, בשם init, שערך ברירת המחדל שלו הוא 0.

הפונקציה תחזיר Generator כך שבכל קריאה ל-next יוחזרו **התוצאות** המצטברות של הפעלת הפונקציה func כפי שיוסבר להלן. האיבר שיוחזר ראשון יהיה הפלט של func על init והאיבר הראשון ב-itr. האיבר השני שיוחזר יהיה הפלט של func על תוצאת החישוב הקודם והאיבר השני ב-itr, וכך הלאה. למשל אם הפונקציה, func, מגדירה כפל ו-init שווה ל-1, אז יוחזרו המכפלות המצטברות של האיברים ב-itr לפי הסדר.

<u>דוגמת ריצה</u>:

```
>>> data = [3, 4, 6, 2, 1]
>>> acc = my_accumulate (data, lambda x, y: x + y)
>>> acc
<generator object my_accumulate at 0x000001130AD98270>
>>> list(acc)
[3, 7, 13, 15, 16]
>>> data = [4, 2, 3]
>>> acc = my_accumulate (data, lambda x, y: x * y, init =1)
>>> next(acc)
4
>>> next(acc)
8
>>> next(acc)
24
>>> next(acc)
StopIteration
```

```
def my_accumulate(itr, func, init=0):
```

בשני הסעיפים הבאים תממשו שני מקרים פרטיים של הפונקציה המובנית zip.
פונקציית zip מקבלת מספר Iterables ומחזירה אובייקט מטיפוס zip שהינו Iterable שמכיל tuples כך שכל אחד מאיברי ה-tuples שייך ל-Iterables אחר בהתאם לסדר בקלט.

<u>לדוגמה</u>:

```
>>>list_str = ["I", "Love", "BGU"]
>>> tuple_int = (1, 2, 3)
>>> list(zip(list_str, tuple_int))
[('I', 1), ('Love', 2), ('BGU', 3)]
>>>list_str = ["one", "two"]
>>> tuple_int = (1, 2, 3)
>>>dict_str = {"First": 1, "second": 2}
>>> zip(list_str, tuple_int, dict_str)
<zip object at 0x00000285AFEE75C0>
>>> tuple(zip(list_str, tuple_int, dict_str))
(('one', 1, 'First'), ('two', 2, 'second'))
```

סעיף ב' (8 נקודות)

ממשו את הפונקציה my_zip(itr_1, itr_2) אשר מקבלת כקלט **שני** Iterables ומחזירה Generator. בכל קריאה ל-next יחזיר גנרטור הפלט את זוג הערכים הבאים כ-tuple, כאשר האיבר הראשון ב-tuple הוא האיבר המתקבל מקריאת ה-next ב-itr_1, והאיבר השני ב-tuple הוא האיבר המתקבל מקריאת ה-next ב-itr_2.

עליכם לעמוד בשני האילוצים הבאים:

- אם אורכי ה-Iterables שונים יש להחזיר tuple לפי ה-Iterables עם מספר האיברים **המינימלי**.
- מותר לעבור על כל איבר בכל Iterable **פעם אחת בלבד**. ניקוד חלקי יינתן למי שלא יעמוד באילוץ הזה.
  <u>תזכורת</u>: Iterable לא בהכרח מממש את הפונקציה len.

<u>דוגמת ריצה</u>:

```
>>> list_1 = ["one", "two", "three"]
>>> list_2 = [1, 2, 3, 4, 5]
>>> var_zip = my_zip(list_1, list_2)
<generator object my_zip at 0x0000027A09093E40>
>>> next(var_zip)
('one', 1)
>>> next(var_zip)
('two', 2)
>>> next(var_zip)
('three', 3)
>>> next(var_zip)
StopIteration
>>> dict_1 = {"1": 1, "2": 2}
>>> set_1 = {1, 2, 3, 4}
>>> var_zip = my_zip(dict_1, set_1)
>>> for elem in var_zip:
>>>     print(elem)
('1', 1)
('2', 2)
```

```
def my_zip(itr_1, itr_2):
```

סעיף ג' (12 נקודות)

כעת עליכם לממש את הפונקציה my_zip_longest שאינה מוגבלת בכמות ה-Iterables ומחזירה Generator שמחזיר tuples כמספר הערכים של ה-Iterable עם מספר האיברים **המקסימלי** מבניהם.

ממשו את הפונקציה my_zip_longest(list_iterators, fillvalue) אשר מקבלת כקלט רשימה המכילה אובייקטים מטיפוס Iterable ומשתנה fillvalue מטיפוס object ומחזירה Generator. בכל קריאה ל-next יחזיר ה-Generator אובייקט מטיפוס tuple באורך מספר ה-Iterables ב-list_iterators , כאשר האיבר הראשון ב-tuple הוא האיבר המתקבל מקריאת ה-next של ה-Iterable במיקום ה-0 ברשימה, האיבר השני ב-tuple הוא האיבר המתקבל מקריאת ה-next של ה-Iterable במיקום ה-1 ברשימה וכך הלאה. כאשר מספר האיברים ב-Iterables המאוחסנים ברשימה אינו זהה, על ה-Generator להחזיר מספר tuples לפי ה-Iterable עם **מספר האיברים המקסימלי**. ערכים חסרים (עבור Iterables קצרים ממנו) יושלמו בערך fillvalue.

<u>שימו לב</u>: גם בסעיף הזה, מעבר על איברי ה-Iterables יותר מפעם אחת יזכה בניקוד חלקי בלבד.

דוגמת ריצה:

```
>>> list_1 = ["one", "two", "three"]
>>> list_2 = [1, 2, 3, 4, 5]
>>> var_zip = my_zip_longest([list_1, list_2], 0)
>>> var_zip
<generator object my_zip_longest at 0x000001F18F9E3E40>
>>> for elem in var_zip:
>>>     print(elem)
('one', 1)
('two', 2)
('three', 3)
(0, 4)
(0, 5)
>>>next(var_zip)
StopIteration
```

```
def my_zip_longest(list_iterators, fillvalue):
```

סעיף ד' (5 נקודות)

נדב ונעם הם שותפים בדירת סטודנטים. כדי לנהל את הוצאות הדירה הם החליטו לקבוע את סכום הכסף שהם מוכנים להוציא על משק הבית בכל אחד מהחודשים. כדי לעקוב אחרי העמידה בתקציב הם החליטו לשמור את הנתונים, התקציב (רשימה) וההוצאות (רשימה של רשימות) לכל חודש, במבנה הנתונים החוזר מהפונקציה my_zip שמימשתם בסעיף ב'.

לדוגמא:

```
>>> limit = [150, 200, 50, 0]
```

התקציב לארבעת החודשים הבאים: 150 ₪ לחודש הראשון, 200 ₪ לחודש השני וכך הלאה.

```
>>> shopping= [[120, 20], [350, 15], [25], [400]]
```

ההוצאות לארבעת החודשים הבאים: 140 ₪ לחודש הראשון, 365 ₪ לחודש השני וכך הלאה.

```
>>>my_zip(limit, shopping)
[(150, [120, 20]), (200, [350, 15]), (50, [25]), (0, [400])]
```

ממשו את הפונקציה use_zip(zip_obj) אשר מקבלת כקלט **Generator החוזר מהפונקציה my_zip**, בשם zip_obj, שמכיל את רשימת ההגבלות ואת רשימת ההוצאות ומחזירה את **סכום** ההוצאות אשר **עמדו בתקציב**.

דגשים: יש לממש את הפתרון **בשורת קוד אחת**. מימוש ביותר משורת קוד אחת יזכה בניקוד חלקי.

דוגמת ריצה:

```
>>> limit = [150, 200, 50, 0]
>>> shopping= [[120, 20], [350, 15], [25], [400]]
>>> var = my_zip(limit, shopping)
>>> use_zip(var)
165
```

הסבר: החודש הראשון (הוצאות של 140 ₪) והחודש השלישי (25 ₪) עומדים במגבלת התקציב.

```
>>> limit = [100, 100, 100, 100]
>>> shopping= [[], [50], [0], [50]]
>>> var = my_zip(limit, shopping)
>>> use_zip(var)
100
```

```
def use_zip(zip_obj):
```

שאלה 2 (34 נקודות)

פוקימונים מיוצרים במפעל של פרופסור אוק ע"י שילוב הקוד הגנטי של שני פוקימונים קיימים. שני הפוקימונים שמעורבים בתהליך מוגדרים כהוריו של הפוקימון החדש שנוצר, שמוגדר כילד של שני הפוקימונים שיצרו אותו. פרופסור אוק מתכנן מערכת מידע חדשה שתעקוב אחרי אילן היוחסין של הפוקימונים אותם הוא מייצר.
בשאלה זו, תממשו שיטות במבנה נתונים מסוג גרף לא מכוון.
לרשותכם מימוש של שתי מחלקות:

1. מימוש מלא של **המחלקה Pokemon**, מייצגת פוקימון ומכילה 3 שדות: שם הפוקימון (מחרוזת), עוצמת התקיפה (מספר לא שלילי) ועוצמת ההגנה (מספר לא שלילי). המחלקה תומכת בשיטות הבאות:
   a. **__eq__(self, other)** – מחזירה True אם שם הפוקימון זהה ו-False אחרת.
   b. **__hash__(self)** – מאפשרת hashing לאובייקט (משמע, ניתן להשתמש במופע של פוקימון כמפתח ב-set וב-dict).
   c. **__repr__(self)** – מחזירה מחרוזת המייצגת את הפוקימון (שם בלבד).
   d. **cross_breeding(self, other, name)** – שיטה המכליאה את הפוקימונים self ו-other ומחזירה פוקימון חדש בשם name, עם עוצמת תקיפה שערכה הוא ממוצע של עוצמות התקיפה, ועוצמת הגנה שערכה הוא ממוצע עוצמות ההגנה של הוריו.

```
class Pokemon:
    def __init__(self, name, a, d):
        self.name = name
        self.a = a
        self.d = d

    def __repr__(self):
        return self.name

    def __eq__(self, other):
        return self.name == other.name

    def __hash__(self):
        return hash(self.name)

    def cross_breeding(self, other, name):
        return Pokemon(name, sum([self.a, other.a])/2, sum([self.d,
other.d])/2)
```

2. מימוש חלקי של המחלקה **PokemonLineageGraph** המייצגת את אילן היוחסין של הפוקימונים באמצעות גרף. גרף אילן היוחסין מיוצג באמצעות שדה בשם pokemon_nodes, המכיל את הפוקימונים כקדקודי הגרף, ושדה בשם pedigree, המכיל את צלעות הגרף בין הורים לילדים. מופע חדש של המחלקה מאותחל כאשר הרשימה של הקדקודים מאותחלת עם מספר פוקימונים שמהווים את הבסיס לשושלת ורשימת צלעות ריקה. **בשאלה תרחיבו את מימוש המחלקה.**

```
class PokemonLineageGraph:
    def __init__(self):
        self.pokemon_nodes = [Pokemon('Pikachu', 5, 5),
                              Pokemon('Charmander', 5, 5),
                              Pokemon('Squirtle', 5, 5),
                              Pokemon('Bulbasaur', 5, 5),
                              Pokemon('Mew', 5, 5)]
        self.pedigree = []

    def __repr__(self):
        return
f"nodes={self.pokemon_nodes}\nedges={self.pedigree}"
```

סעיף א' (5 נקודות)

צלעות הגרף (pedigree) ממומשות כרשימה של tuples המחזיקים זוגות קודקודים (מופעים של פוקימונים). מכיוון שהגרף אינו כיווני, צלע בין פוקימון A לפוקימון B תופיע פעמיים ברשימת צלעות הגרף: צלע בין A ל-B (A, B) וצלע בין B ל-A (B, A).

ממשו את השיטה add_pokemon(self, poke_name, ancestors) אשר מקבלת כקלט שני ארגומנטים:

- שם הפוקימון החדש (poke_name, מחרוזת).
- זוג אבות של הפוקימון החדש כ-Tuple המכיל שני פוקימונים (ancestors).

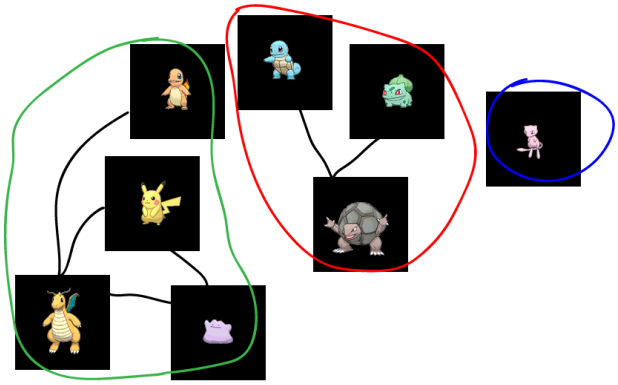
השיטה מוסיפה פוקימון חדש לגרף, וצלעות המחברות את הפוקימון החדש להוריו.

<u>דוגמת ריצה:</u>

פרופ' אוק ייצר את הפוקימון החדש Zubat בעזרת הפוקימונים Charmander ו-Squirtle:

```
>>> poke_lineage = PokemonLineageGraph()

>>> poke_lineage .add_pokemon('Zubat', (poke_lineage .nodes[1], poke_lineage .nodes[2]))

>>> print(poke_lineage)

nodes=[Pikachu, Charmander, Squirtle, Bulbasaur, Mew, Zubat]

edges=[(Charmander, Zubat), (Squirtle, Zubat), (Zubat, Charmander), (Zubat, Squirtle)]
```

```
def add_pokemon(self, poke_name, ancestors):
```



סעיף ב' (11 נקודות)

קבוצה של פוקימונים מוגדרת כ-"**משפחה**" כאשר קיים מסלול בין כל זוג פוקימונים בקבוצה. באיור המצורף, כל קבוצת פוקימונים המוקפת בצבע היא משפחה נפרדת של פוקימונים. ממשו את השיטה get_pokemon_families (self) אשר מחזירה רשימה המכילה לכל משפחת פוקימונים בגרף רשימה של מצביעים לכל הפוקימונים במשפחה. אין חשיבות לסדר המשפחות בקבוצה. אין צורך לבצע העתקה של הפוקימונים.

<u>דוגמאות ריצה:</u>

1. Dragonite נוצר מ-Pikachu ומ-Charmander, ו-Ditto נוצר מ-Pikachu ומ-Drgonite. קיימות 4 משפחות פוקימונים בגרף הנוכחי. שלוש קבוצות המורכבות מפוקימון בסיס יחיד וקבוצה המכילה את Dragonite, Pikachu, Ditto ו-Charmander.

```
>>> poke_lineage = PokemonLineageGraph()

>>> poke_lineage.add_pokemon('Dragonite', (poke_lineage.nodes[0], poke_lineage.nodes[1]))
>>> poke_lineage.add_pokemon('Ditto', (poke_lineage.nodes[-1], poke_lineage.nodes[0]))

>>> poke_lineage.get_pokemon_families()

[[Pikachu, Ditto, Dragonite, Charmander], [Squirtle], [Bulbasaur], [Mew]]
```

2. בהמשך לדוגמה הקודמת, פרופ' אוק יצר את הפוקימון Golem מהפוקימונים Squirtle ו-Balbasaur. כעת קיימות בסה"כ 3 משפחות פוקימונים (מתואר באיור בתחילת השאלה).

```
>>> poke_lineage = PokemonLineageGraph()

>>>poke_lineage.add_pokemon('Dragonite',(poke_lineage.nodes[0],   poke_lineage.nodes[1]))
>>> poke_lineage.add_pokemon('Ditto', (poke_lineage.nodes[-1], poke_lineage.nodes[0]))

>>> poke_lineage.add_pokemon('Golem', (poke_lineage.nodes[2], poke_lineage.nodes[3]))

>>> poke_lineage.get_pokemon_families()

[[Pikachu, Ditto, Dragonite, Charmander], [Squirtle, Golem, Bulbasaur], [Mew]]
```

```
def get_pokemon_families (self):
```

סעיף ג' (3 נקודות)

ממשו את השיטה get_pokemon_families_strengths(self) אשר מחזירה רשימה המכילה לכל משפחת פוקימונים בגרף את **עוצמת התקיפה המצטברת** שמוגדרת כסכום עוצמות התקיפה של כל הפוקימונים במשפחה. אין חשיבות לסדר המשפחות בקבוצה.

<u>דוגמאות ריצה, בהתאם לשתי הדוגמאות בסעיף הקודם:</u>

1. קיימות שלוש משפחות פוקימונים המורכבות מפוקימון יחיד עם עוצמת תקיפה מצטברת של 5 לכ"א, וקבוצה עם ארבעה פוקימונים עם עוצמת תקיפה מצטברת של 20.

```
>>> poke_lineage.get_pokemon_families_strengths()
[20, 5, 5, 5]
```

2. שלוש משפחות פוקימונים, ראו איור

```
>>> poke_lineage.get_pokemon_families_strengths()
[20, 15, 5]
```

```
def get_pokemon_families_strengths(self):
```

סעיף ד' (6 נקודות)

קרובי משפחה מיידים של פוקימון מוגדרים להיות קבוצת ההורים והילדים שלו. ממשו את השיטה get_poke_relatives(self, poke) אשר מקבלת מופע של פוקימון (poke) ומחזירה קבוצה (set) המכילה את קרובי המשפחה המיידים שלו.

<u>דוגמת ריצה:</u>
בדוגמה הבאה Dragonite נוצר מPikachu ו-Charmander, ו-Ditto נוצר מ-Pikachu ומ-Drgonite. קבוצת ההורים והילדים של Dragonite כוללת את הפוקימונים Pikachu, Charmander ו-Ditto.

```
>>> poke_lineage = PokemonLineageGraph()

>>> poke_lineage.add_pokemon('Dragonite', (poke_lineage.nodes[0], poke_lineage.nodes[1]))
>>> poke_lineage.add_pokemon('Ditto', (poke_lineage.nodes[-1], poke_lineage.nodes[0]))

>>> print(poke_lineage.get_poke_relatives(Pokemon('Dragonite', 0, 0)))

{Ditto, Pikachu, Charmander}
```

```
def get_poke_relatives(self, poke):
```

סעיף ה' (8 נקודות)

רמת "משפחתיות" (kinship) של פוקימון מוגדרת כמספר קרובי המשפחה המיידיים שלו. לדוגמה, רמת המשפחתיות של Dragonite (הפוקימון השמאלי התחתון במשפחה הירוקה באיור) היא 3.
ממשו את השיטה sort_families_by_max_kinship(self) המחזירה רשימה **ממוינת** של משפחות הפוקימונים ע"פ רמת משפחתיות עולה שנקבעת לפי הפוקימון עם רמת המשפחתיות המקסימלית בכל משפחה.

לדוגמה, הפוקימון עם רמת המשפחתיות המקסימלית במשפחה הירוקה באיור הוא Dragonite עם רמת משפחתיות של 3, הפוקימון Golem הוא המקסימלי במשפחה האדומה עם רמת משפחתיות של 2, והפוקימון Mew הוא המקסימלי במשפחה הכחולה עם רמת משפחתיות של 0. לכן, השיטה תחזיר רשימה המכילה את המשפחות לפי הסדר: כחולה (אינדקס 0), אדומה, ירוקה. ראו דוגמת ריצה:

```
>>>sorted_families=poke_lineage.sort_families_by_max_kinship()

>>> print(sorted_families)

[[Mew], [Squirtle, Golem, Bulbasaur], [Pikachu, Dragonite, Charmander, Ditto]]
```

```
def sort_families_by_max_kinship(self):
```

שאלה 3 (40 נקודות)

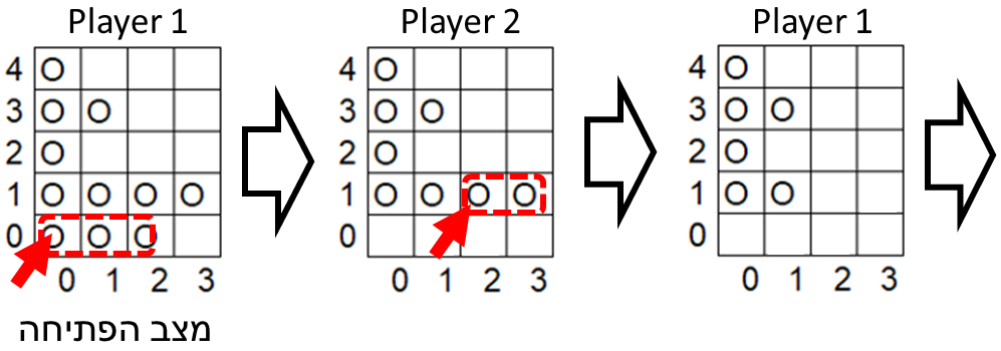
בשאלה זאת נממש משחק בשם "האחרון בשורה". המשחק מורכב מלוח מלבני ואבני משחק ומיועד לשני שחקנים. לוח המשחק יכיל רצפים של אבני משחק בשורות הלוח משמאל לימין. מטרת המשחק היא לא להסיר את האבן האחרונה מהלוח, כלומר השחקן המפסיד הוא זה שמסיר את האבן האחרונה.

מהלך המשחק:

כל שחקן בתורו מסיר אבנים מהלוח לפי הכללים הבאים:

1. השחקן מסיר אבני משחק (לפחות אחת) מהשורה התחתונה ביותר בה קיימות אבני משחק. לדוגמא, במצב הפתיחה באיור (לוח המשחק השמאלי ביותר) ניתן להסיר אבני משחק רק משורה 0.
2. השחקן מסיר רצף אבנים מאינדקס מסוים ועד לסוף השורה לימין. לדוגמא, במצב הפתיחה באיור קיימים שלושה מהלכים אפשריים: ניתן להסיר את האבן באינדקס 2 (החל באינדקס 2), או את האבנים באינדקס 2 ו-1 (החל מאינדקס 1), או את האבנים באינדקס 2, 1, 0 (החל מאינדקס 0).

דוגמא: מהלך משחק אפשרי משמאל (מצב הפתיחה) לימין. החץ מסמן את המהלך שנבחר, כלומר המיקום של אבן המשחק ממנה וימינה יוסרו האבנים (מסומנים באמצעות מלבן מקווקו).



ייצוג לוח המשחק: נייצג את המצב הנוכחי של לוח משחק בגודל n x m באמצעות רשימה באורך n (מספר השורות), כאשר האינדקס ה-i מכיל את מספר האבנים בשורה ה-i. נכנה מצב נוכחי של לוח משחק בשם "**קונפיגורציה**".

לדוגמא, לוח המשחק באיור ייוצג באמצעות הקונפיגורציות הבאות (משמאל לימין):

[3,4,1,2,1] → [0,4,1,2,1] → [0,2,1,2,1]

סעיף א' (4 נקודות)

ממשו את הפונקציה current_row(board) אשר מקבלת כקלט קונפיגורציה ומחזירה את מספר השורה שממנה יוסרו אבנים במהלך הבא. אם אין שורה כזו - החזירו None.

דוגמאות ריצה:

```
>>> current_row([2, 8, 8, 8, 8])
0
>>> current_row([0, 0, 0])
None
>>> current_row([0, 0, 3, 8, 8, 8, 8])
2
```

```
def current_row(board):
```

סעיף ב' (4 נקודות)

נייצג **מהלך** באמצעות האינדקס שממנו מסירים אבנים עד לסוף השורה. לדוגמא, עבור הקונפיגורציה שמתארת את מצב הפתיחה באיור, המהלכים האפשריים הם 2 (הסרה של אבן אחת בשורה 0 ובאינדקס 2), 1 (הסרה של שתי אבנים), או 0 (הסרה של השורה כולה).

ממשו את הפונקציה do_move(board, move) המקבלת כקלט קונפיגורציה ומהלך ומחזירה את הקונפיגורציה לאחר ביצוע המהלך.

<u>דוגמאות ריצה:</u>

```
>>> do_move([5, 5, 5], 0)
[0, 5, 5]
>>> do_move([3, 5, 5], 1)
[1, 5, 5]
>>> do_move([0, 5, 5], 4)
[0, 4, 5]
```

```
def do_move(board, move):
```

סעיף ג' (14 נקודות)

בסעיף זה תעברו על כל תרחישי המשחק האפשריים ותספרו בכמה מהם ניצח השחקן הראשון ובכמה ניצח השני. תרחיש משחק מוגדר כסדרה ***ייחודית*** של מהלכים עד לניצחון של אחד השחקנים.

ממשו את **הפונקציה** play(board) המקבלת קונפיגורציה ומחזירה רשימה באורך 2. האיבר הראשון (אינדקס 0) ברשימת הפלט יחזיק את מספר התרחישים בהם ניצח השחקן הראשון. האיבר השני יחזיק את מספר התרחישים בהם ניצח השחקן השני.

```
def play(board):
```

סעיף ד' (10 נקודות)

כפי שראינו בהרצאה, **קונפיגורציה מפסידה** היא קונפיגורציה ממנה כל מהלך חוקי יוביל להפסד במשחק. כלומר השחקן שנמצא בקונפיגורציה המפסידה, יפסיד ללא תלות במהלכים שיבחר לשחק, אם השחקן השני ישחק בצורה אופטימלית.

דוגמאות פשוטות לקונפיגורציות מפסידות הן [0,0,0,0,1] בה נשארה רק האבן האחרונה, או הקונפיגורציה [1,1,1] שבה כל שחקן בתורו מחויב להסיר את השורה הבאה.

קונפיגורציה מנצחת היא קונפיגורציה שניתן להגיע ממנה במהלך אחד לקונפיגורציה מפסידה. כלומר קיים מהלך מנצח לשחקן שתורו לשחק. לדוגמא, הקונפיגורציה [3,1,1] היא קונפיגורציה מנצחת באמצעות המהלך המנצח 1 (כלומר בחירת האינדקס 1 שגורר הסרה של שתי אבנים בשורה ה-0) שמוביל לקונפיגורציה המפסידה [1,1,1].

דוגמא נוספת, הקונפיגורציה [2,2] היא קונפיגורציה מנצחת באמצעות המהלך המנצח 1 שמובילה לקונפיגורציה המפסידה [1,2] (משום שהמשך משחק יעבור דרך הקונפיגורציות [0,2] ואז המהלך שיוביל ל-[0,1] ולניצחון במהלך הבא).

ממשו את הפונקציה win(board) המקבלת קונפיגורציה ומחזירה האם היא קונפיגורציה מנצחת.

דוגמאות ריצה:

```
>>> win([0, 0, 0, 0, 1])
False
>>> win([1, 1,1])
False
>>> win([3, 1,1])
True
>>> win([2, 2])
True
```

```
def win(board):
```

סעיף ה' (8 נקודות)

סיבוכיות זמן הריצה של סעיף ד' הוא אקספוננציאלי בגודל הקלט. איך ניתן לייעל סיבוכיות זו? הסבירו במילים את המימוש ומדוע הוא יוביל לקוד הרבה יותר יעיל. תשובה ללא נימוק מדויק תזכה בחלק קטן בלבד מהנקודות.

הקונפיגורציה ההתחלתית של לוח משחק מלא בגודל n x m היא [m,m,…,m]. מה תהיה סיבוכיות זמן הריצה של הפתרון שהצעתם עבור לוח משחק התחלתי מלא בגודל n x m? נמקו.