מס' ת"ז_____ מס' מחברת_____

מבחן בקורס מבוא מורחב למדעי המחשב, CS1001.py

עם תיקונים קלים, בעקבות הבהרות שניתנו בזמן הבחינה

סמסטר ב' 2014

מועד ב, 11/8/2014

מרצים: פרופ' בני שור, אמיר רובינשטיין

מתרגלים: מיכל קליינבורט, יעל ברן

משך הבחינה: 3 שעות.

חומר עזר מותר: 2 דפי עזר (דו צדדיים) בגודל A4 כ"א.

- במבחן 15 עמודים – בידקו שכולם בידיכם. בנוסף, בסוף הבחינה ישנו דף נוסף ריק, לשימוש במקרה חירום בלבד.
- יש לכתוב את כל התשובות בטופס הבחינה. המחברת תשמש כטיוטה בלבד ולא תיבדק.
- יש לענות על כל ארבע השאלות.
- בכל השאלות, אלא אם נכתב במפורש אחרת:
  - אם עליכם לכתוב פונקציה, אין צורך לבדוק בה את תקינות הקלט שלה
  - מותר להסתמך על סעיפים קודמים, גם אם לא עניתם עליהם
  - ניתן לצטט טענות שנטענו בהרצאה או בשיעורי התרגול
- אנו ממליצים לא "להיתקע" על אף שאלה, אלא להמשיך לשאלות אחרות ולחזור לשאלה אח"כ.
- בכל סעיף בשאלות הפתוחות ניתן לכתוב "איני יודע/ת" ולא לכתוב שום טקסט נוסף. במקרה זה יינתן 20% מציון הסעיף.
- יש לכתוב את כל התשובות במקום המוקצב ובכתב קריא. חריגות משמעותיות מהמקום המוקצב, או תשובות הכתובות בכתב קטן מדי, לא ייקראו; תשובות שדורשות מאמצים רבים להבנתן עלולות לגרור הורדת ציון.

בהצלחה !

טבלת ציונים: (לשימוש הבודקים)

שאלה	ערך	ניקוד
1	25	
2	25	
3	30	
4	20	
סה"כ	100	

שאלה 1 (25 נק')

המחלקה Matrix בה השתמשנו רבות בקורס, מייצגת מטריצות: אם M מטריצה, ו-()M.dim שווה (n,m), אז בייצוג הפנימי של M יש n רשימות, כל אחת בגודל m, ובסה"כ n x m תאים, שכ"א מכיל מספר. ל-n, m שהם בסדרי גודל של עשרות אלפים, מטריצה כזו מהווה מעמסה לא זניחה על זיכרון המחשב.

יש הֶקשרים רבים בהם מספר הכניסות השונות מאפס במטריצה הוא רק קבוע כפול n+m. במקרה כזה אנו אומרים כי המטריצה היא **דלילה**. במקרה זה נרצה להגדיר מחלקה SparseMatrix, המייצגת מטריצה תוך שימוש בזיכרון פרופורציוני למספר האיברים השונים מאפס (אנו מניחים כי ייצוג של מספר צורך O(1) זיכרון).

המחלקה המתוארת כאן מגדירה מטריצה דלילה באמצעות מילון (dictionary) המאותחל כמילון ריק. מפתחות המילון הם זוגות (i,j) בתחום המתאים. מצ"ב המתודות __init__, __repr__, dim, ו-__getitem__.

```
class SparseMatrix:
  """
  Represents a sparse rectangular matrix with n rows and m columns.
  """

  def __init__(self, n, m):
    """
    Create a sparse n-by-m matrix of values.
    Inner representation: dictionary.
    """
    assert n > 0 and m > 0
    self.rows = n
    self.columns = m
    self.entries={}

  def dim(self):
    return self.rows, self.columns

  def __repr__(self):
    return "<SparseMatrix {}>".format(self.entries)

  def __getitem__(self, ij): #ij is a tuple (i,j).
                             #Allows m[i,j] instead m[i][j]
    i,j = ij
    if isinstance(i, int) and isinstance(j, int):
        if (i,j) in self.entries:
            return self.entries[(i,j)]
        elif i<0 or i>=self.rows or j<0 or j>=self.columns:
            return "index out of bounds"
        else:
            return 0
```

בסעיפים הבאים תוכלו להניח כי פעולות פשוטות על מילון דורשות O(1) זמן בממוצע: key in diction (בדיקה האם מפתח נמצא במילון), diction[key] (שליפת ערך אם המפתח נמצא), diction[key]=val (השמת ערך למילון), del diction[key] (מחיקת מפתח הנמצא במילון יחד עם הערך המתאים). הפעולה for key in diction (מעבר על כל המפתחות במילון) דורשת O(k) זמן בממוצע, כאשר k הוא מספר המפתחות במילון.

א. (5 נק') השלימו את הקוד של המתודה __setitem__ המציבה ערך val לכניסה [i,j] במטריצה. שימו לב כי יש לטפל באופן שונה במקרה שהערך שונה מ-0 ובמקרה בו הערך שווה 0. המתודה אינה מחזירה שום איבר, אך משנה את המטריצה הדלילה. המתודה צריכה לפעול בזמן $O(1)$ בממוצע.

מצ"ב דוגמת הרצה:

```
>>> M=SparseMatrix(3,4)
>>> M[(2,3)]=13
>>> M
<SparseMatrix {(2, 3): 13}>
>>> M[(2,3)]=0
>>> M
<SparseMatrix {}>
```

```
def __setitem__(self, ij, val): #ij is a tuple (i,j).
                                    #Allows m[i,j] instead m[i][j]
  i,j = ij
  if isinstance(i, int) and isinstance(j, int):
      assert isinstance(val, (int, float))
  if i<0 or i>=self.rows or j<0 or j>=self.columns:
      print("index out of bounds")
```

ב. (10 נק') השלימו את הקוד של המתודה __add__ המחברת שתי מטריצות בעלות אותם מימדים. המתודה מחזירה מטריצה דלילה חדשה, שהיא סכום שתי המטריצות (אם מטריצות הקלט הן N ו- M ומטריצת התוצאה היא R, אז לכל זוג אינדקסים חוקיים i,j מתקיים כי R[i,j] שווה לסכום M[i,j] +N[i,j]). המתודה צריכה לפעול בזמן O(c+d) בממוצע, כאשר במטריצות הקלט יש c, d איברים שונים מאפס, בהתאמה.

מצ"ב דוגמת הרצה:

```
>>> M=SparseMatrix(3,4)
>>> N=SparseMatrix(3,4)
>>> M[2,3]=10
>>> M[1,3]=5
>>> N[2,3]=-10
>>> N[1,3]=4
>>> N[0,3]=2
>>> N+M
<SparseMatrix {(0, 3): 2, (1, 3): 9}>
```

```
def __add__(self, other):
  assert self.dim()== other.dim()
  n,m = self.dim()
  new = SparseMatrix(n,m)
```

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

```
  return new
```

ג. (10 נק') אנו אומרים כי זוג נקודות M[i,j] ו- M[i', j'] נמצאות על אותו אלכסון, אם ערכי שתיהן שונים מ-0, וכן $i - j = i' - j'$. במקרה זה נאמר כי זוג הנקודות נמצא על האלכסון $d = i - j$ (שימו לב כי d יכול להיות שלילי).
השלימו את הקוד של המתודה diagonal המקבלת כקלט מטריצה דלילה ומחזירה כפלט את האלכסון $d = i - j$ המכיל מספר **מקסימלי** של נקודות שונות מ-0 מהמטריצה M. אם יש יותר מאלכסון אחד כזה, המתודה תחזיר אחד מהם. המתודה צריכה לעבוד בזמן O(k) בממוצע, כאשר k הוא מספר הכניסות השונות מ-0 במטריצת הקלט.

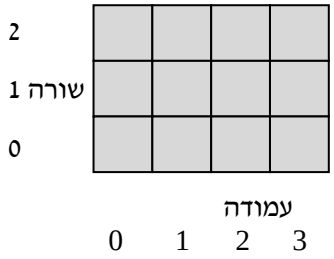
ממשו את הפונקציה בפייתון (במקום המוקצב למטה).

```
def diagomal(self):
  diagonals = {}
  for key in self.entries:
```

שאלה 2 (25 נק')

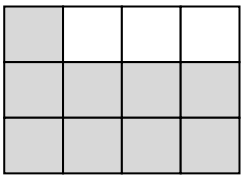
בשיעור (הרצאה 14) ובשיעורי הבית (מטלה 4) הוצג המשחק זלול! (Munch!), משחק לוח לשני שחקנים אשר גורעים (זוללים) בזה אחר זה קוביות מתוך טבלת שוקולד על פי חוקים מוסכמים מראש. בשאלה זו נדון בגרסא "מרוככת" של המשחק. בגרסא זו, כל שחקן חייב לבחור בתורו קוביית שוקולד קיימת (שלא נאכלה עדין) הנמצאת או בשורה התחתונה או בעמודה השמאלית של טבלת השוקולד. זה מבטיח כי הטבלה הנותרת לאחר הצעד היא מלבנית (אין בה "מדרגות"). הקובייה בשורה התחתונה ובעמודה השמאלית היא, כרגיל, מורעלת, והשחקן שאוכל אותה מפסיד.

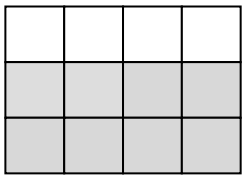
זה מאפשר לייצג קונפיגורציה נתונה באופן קומפקטי ע"י מספר השורות ומספר העמודות. לדוגמה, הייצוג הקומפקטי של מצב המשחק ההתחלתי של טבלה בגודל $m=4, n=3$ הוא [3,4] וכך נראה הלוח:



[3,4]

במשחק ה"מרוכך" הטבלה מימין, למשל, אינה המשך חוקי של [3,4], אך הטבלה משמאל מהווה המשך חוקי של אותה קונפיגורציה.





אנו מציגים למטה מימוש לפונקציה win(n,m) אשר מקבלת מצב לוח חוקי ברגע נתון של המשחק (קונפיגורציה, ע"פ המינוח למעלה), 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 שנותרה בו רק הקוביה האחרונה, הפונקציה תחזיר False (כי השחקן שתורו לשחק חייב לבלוע את הקוביה המורעלת הזו). מצב הלוח הנוכחי מועבר לפונקציה בייצוג קומפקטי, ע"י זוג הטבעיים החיוביים n,m.

להלן קוד רקורסיבי "טהור" לחישוב הפונקציה win(n,m):

```
def win(n,m):
     assert n>0 and m>0
     if n==m==1:
          return False
     for i in range(1,n):
          if not win(i,m):
               return True
     for j in range(1,m):
          if not win(n,j):
               return True
     return False
```

א. (10 נק') שנו את הקוד הרקורסיבי ה"טהור" הנ"ל לקוד רקורסיבי המשתמש ב-memoization, ומחשב את אותה פונקציה כמו win(n,m) עבור המשחק זלול! המרוכך. הקוד החדש צריך למזער את מספר הקריאות הרקורסיביות (כלומר להשתמש בערכים שנמצאים כבר במילון בכל מקרה שזה אפשרי). החתימה של הקוד החדש צריכה להיות win_mem(n,m,win_dict), כאשר win_dict הוא מילון בו מאוחסנות התוצאות שחושבו כבר. הקריאה לפונקציה ברמה העליונה צריכה להיות מהצורה win_mem(n,m,{(1,1):False}).

הסבירו תחילה במילים מה עושה הקוד שלכם:

השלימו כעת את שורות הקוד הבאות לקוד פייתון שלם:

```
def  win_mem(n,m,win_dict):
      assert n>0 and m>0
      if (n,m) in win_dict:
            return win_dict[(n,m)]
      else:
```

ב. (10 נק') ציירו את עץ הרקורסיה המתקבל מהרצת הקוד שהשלמתם בסעיף א', עבור טבלת השוקולד בגודל m=4, n=3 (שורש העץ (3,4) כבר מופיע). יש לציין רק את הצמתים עליהם הקוד שלכם קורא לפונקציה win_mem. בכל צומת, ציינו רק את מספר השורות ומספר העמודות, (n,m). סמנו ע"י W כל קונפיגורציה מנצחת, וע"י L כל קונפיגורציה מפסידה. ציינו גם כמה פעמים מופיעה הקונפיגורציה (2,1) בעץ שלכם.

(3,4)

מספר הקונפיגורציות (2,1) בעץ: _____

ג. (5 נק') מהו מספר הקריאות ל- win_mem על הקלט (n, m,{(1,1):False})?
(מספר הקריאות הרקורסיביות, כולל הקריאה ברמה העליונה). כתבו את הביטוי המפורש וספקו הסבר קצר.

שאלה 3 (30 נק')

בהינתן שתי מחרוזות s1 ו- s2, אנו מעוניינים לחשב את החפיפה הארוכה ביותר של סיפא (סיומת) של s1 ורישא (תחילית) של s2. נסמן ב- k את אורכה של החפיפה המקסימלית הנ"ל. נניח כי שתי המחרוזות באותו אורך (אין צורך לוודא זאת), ונסמנו ע"י n.

למשל, עבור "abcbc" = s1 ו- "bcbcd" = s2, k הגדול ביותר כנ"ל הוא k=4 (שמתאים למחרוזת "bcbc"), ואילו עבור "bcd" = s1 ו- "abc"=s2, k הגדול ביותר הוא 0 (מתאים למחרוזת הריקה).

א. (5 נק') ציינו האם הטענה הבאה נכונה או לא: אם הסיפא באורך k של s1 והרישא באורך k של s2 אינן שוות, אז גם הסיפא באורך k+1 של s1 והרישא באורך k+1 של s2 אינן שוות. אם לדעתכם הטענה נכונה – הסבירו. אחרת, תנו דוגמה נגדית.

ב. (5 נק') להלן אלגוריתם נאיבי למציאת החפיפה המקסימלית הנ"ל:

```
def find_overlap_naive(s1, s2):
    overlap = ""
    for i in range(len(s1)+1):
        if s1[-i:] == s2[:i]:
            overlap = s1[-i:]

    return overlap
```

מהי סיבוכיות הזמן של פתרון זה, כתלות באורך המחרוזות n? תנו תשובה הדוקה ככל שתוכלו במונחי סדר גודל אסימפטוטיים, תוך שימוש בסימון O(…).

O(_____)

בסעיפים ג'-ה' עליכם להשלים פתרון לבעייה, המתבסס על הרעיון של אלגוריתם Karp-Rabin. לנוחיותכם, מופיע בהמשך הקוד של האלגוריתם שנלמד בכיתה (מותר להיעזר בו בכל דרך שתרצו). ערכי ברירת המחדל של basis ושל r יישארו כפי שראינו בכיתה, ואין לשנות אותם. לצורכי ניתוח סיבוכיות נניח, כי כל פעולה אריתמטית (כגון כפל, חזקה, מודולו וכו') רצה בזמן $O(1)$.

ג. (5 נק') השלימו את הפונקציה suff_fingerprint המקבלת מחרוזת string ומחזירה רשימה של טביעות האצבע (fingerprints) של כל הסיפות (סיומות) של string, בסיבוכיות זמן $O(n)$.

```
def suff_fingerprint(string, basis=2**16, r=2**32-3):
    f = [ ord(string[-1]) % r ]
    power = 1

    for _____:

        _____

        _____

        list.append(f,new_fingerprint)#append f[s] to existing f

    return f
```

ד. (5 נק') השלימו את הפונקציה pre_fingerprint המקבלת מחרוזת string ומחזירה רשימה של טביעות האצבע (fingerprints) של כל הרישות (תחיליות) של string, בסיבוכיות זמן $O(n)$.

```
def pre_fingerprint(string, basis=2**16, r=2**32-3):
    f = [ ord(string[0]) % r ]

    for _____:

        _____

        list.append(f,new_fingerprint)#append f[s] to existing f

    return f
```

ה. (10 נק') השלימו את הפונקציה find_overlap המקבלת שתי מחרוזות s1 ו- s2 באותו אורך n (אין צורך לוודא זאת), ומחזירה את החפיפה הארוכה ביותר בין סיפא (סיומת) של s1 ורישה (תחילית) של s2. הפונקציה תסתמך על השוואת טביעות אצבע שחושבו באמצעות הפונקציות משני הסעיפים הקודמים. אם הפונקציה תחזיר תוצאה שגויה בשל כך - אין צורך לבדוק זאת. תוצאות שגויות מסיבות אחרות תחשבנה כטעות. סיבוכיות הזמן הדרושה היא $O(n)$.

שימו לב: יש להשתמש בפונקציות משני הסעיפים הקודמים גם אם לא פתרתם אותם נכון. פתרון אחר לא יתקבל.

```
def find_overlap(s1, s2, basis=2**16, r=2**32-3):
```

לנוכחיותכם, הקוד מהכיתה:

```
def fingerprint(string, basis=2**16, r=2**32-3):
    """ used to compute karp-rabin fingerprint of the pattern
    employs Horner method (modulo r) """
    partial_sum = 0
    for x in string:
        partial_sum = (partial_sum*basis+ord(x)) % r
    return partial_sum


def text_fingerprint(string, length, basis=2**16, r=2**32-3):
    """ used to computes karp-rabin fingerprint of the text """
    f = []
    b_power = pow(basis,length-1,r)
    list.append(f, fingerprint(string[0:length], basis, r))
    # f[0] equals first text fingerprint
    for s in range(1,len(string)-length+1):
        new_fingerprint =((f[s-1]-ord(string[s-1])*b_power)*basis
                          +ord(string[s+length-1])) % r
            # compute f[s], based on f[s-1]
        list.append(f,new_fingerprint)# append f[s] to existing f
    return f


def find_matches_KR(pattern, text, basis=2**16, r=2**32-3):
    if len(pattern) > len(text):
        return []
    p = fingerprint(pattern,basis,r)
    f = text_fingerprint(text,len(pattern),basis,r)
    matches = [s for s, f_s in enumerate(f) if f_s == p]
    return matches
```

שאלה 4 (20 נק')

בשאלה זו אנו עוסקים בדחיסת מחרוזות. יחס הדחיסה של מחרוזת מוגדר כיחס בין מספר הביטים הדרוש על מנת לייצג את המחרוזת הדחוסה לבין מספר הביטים הדרוש על מנת לייצג את המחרוזת המקורית (כלומר מספר תווי ה-ascii במחרוזת כפול 7).

בתרגול הוצגה הפונקציה compression_ratio(text, corpus) שמחזירה את יחס הדחיסה של המחרוזת text לפי קוד האפמן, שנבנה על פי המחרוזת corpus כקורפוס:

```
def compression_ratio(text, corpus):
  d_code = generate_code(build_huffman_tree(char_count(corpus)))
  len_compress = len(compress(text, d_code))
  len_unicode = len(text)*7
  return len_compress/len_unicode
```

א. (8 נק') תהי s מחרוזת נתונה. נחשב את הערכים הבאים:

```
a = compression_ratio(s,s)
b = compression_ratio(s[::-1],s)
```

סמנו איזה מבין המשפטים הבאים נכון בהכרח, והסבירו:

1. a == b
2. הדחיסה של המחרוזת s על פי הקורפוס s היא אופטימלית ובהכרח מתקיים ש: a < b.
3. הדחיסה של המחרוזת s[::-1] על פי הקורפוס s היא אופטימלית ובהכרח מתקיים ש: b < a.
4. לא ניתן לקבוע בוודאות מה היחס בין a ל-b, זה תלוי במחרוזת s.

הפונקציה lz_ratio(text) הוצגה גם היא בתרגול. פונקציה זו מחזירה את יחס הדחיסה של המחרוזת text לפי אלגוריתם למפל-זיו לדחיסת מחרוזות (bits, הפרמטר השני שמחזירה process הוא המחרוזת הדחוסה):

```
def lz_ratio(text):
    inter1, bits, inter2, text2 = process(text)
    return len(bits)/(len(text)*7)
```

ב. (6 נק') תנו דוגמא למחרוזת s באורך שבין 4 ל- 10 תווים, עבורה מתקיים ש:

`lz_ratio(s) == lz_ratio(s[::-1])` וגם `s != s[::-1]` (המחרוזת איננה פלינדרום).

ג. (6 נק') תנו דוגמא למחרוזת s באורך שבין 4 ל- 10 תווים, עבורה מתקיים ש:

`lz_ratio(s) != lz_ratio(s[::-1])`.

סוף!

דף נוסף למקרה הצורך