אוניברסיטת תל אביב - בית הספר למדעי המחשב
מבוא מורחב למדעי המחשב, חורף 2025-26

תרגיל בית מספר 4 - להגשה עד 28.12.2025 בשעה 23:59

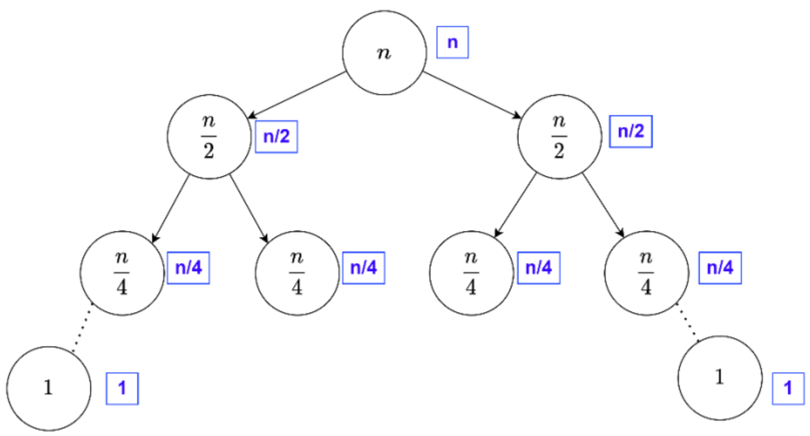
קראו בעיון את הנחיות העבודה וההגשה המופיעות באתר הקורס, תחת התיקייה assignments.
חריגה מההנחיות תגרור ירידת ציון / פסילת התרגיל.

הנחיות לצורת ההגשה:
- תשובותיכם יוגשו בקובץ pdf ובקובץ py בהתאם להנחיות בכל שאלה.
- 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.
- השתמשו בקובץ השלד skeleton4.py כבסיס לקובץ אותו אתם מגישים.
- לא לשכוח לשנות את שם הקובץ למספר ת״ז שלכם לפני ההגשה, עם סיומת py.

הנחיות לפתרון:
- הקפידו לענות על כל מה שנשאלתם.
- בכל שאלה, אלא אם מצוין אחרת באופן מפורש, ניתן להניח כי הקלט תקין.
- אין להשתמש בספריות חיצוניות פרט לספריות math, random, time אלא אם נאמר במפורש אחרת.
- תשובות מילוליות והסברים צריכים להיות תמציתיים, קולעים וברורים.
  להנחיה זו מטרה כפולה:
  i. על מנת שנוכל לבדוק את התרגילים שלכם בזמן סביר.
  ii. 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.
- כיוון שלמדנו בשבועות האחרונים כיצד לנתח את זמן הריצה של הקוד שלנו, החל מתרגיל זה ולאורך שארית הסמסטר (וכן במבחן) נדרוש שכל הפונקציות שאנו ממשים תהיינה יעילות ככל הניתן. לדוגמה, אם ניתן לממש פתרון לבעיה בסיבוכיות $O(logn)$, ואתם מימשתם פתרון בסיבוכיות $\Theta(n)$, תקבלו ניקוד חלקי על הפתרון.
- בשאלות שבהן ישנה דרישה לניתוח סיבוכיות זמן הריצה, הכוונה היא לסיבוכיות זמן הריצה של המקרה הגרוע ביותר (worst-case complexity). כמו כן, אלא אם כן צוין אחרת, ניתן להגיש פתרונות שרצים בזמן יעיל יותר מהדרישה בתרגיל. (לדוגמא, אם נדרש שסיבוכיות הזמן של הפתרון תהיה $O(n^2)$, ניתן להגיש קוד שסיבוכיות זמן הריצה שלו היא $O(n)$).

הערות כלליות לתרגיל זה ולתרגילים הבאים:

כאשר אתם מתבקשים לצייר עץ רקורסיה, מומלץ להיעזר בכלים דיגיטליים כמו draw.io כדי לצייר את העץ. עם זאת, ניתן לצייר את עצי הרקורסיה בכתב יד ולסרוק, כל עוד הציור ברור לחלוטין. **ציור שאינו ברור לא ייבדק.**
בציורכם הקפידו על הדברים הבאים: כל צומת בעץ מייצג קריאה רקורסיבית – בתוך הצומת כתבו את הקלט, או את אורך הקלט, המתאים לצומת זה. לצד כל צומת ניתן לכתוב את כמות העבודה שמתבצעת בצומת (ניתן גם לפרט מתחת לציור). לדוגמה, את עץ הרקורסיה עבור המקרה הטוב ביותר של Quicksort (כפי שראיתם בהרצאה) נצייר באופן הבא:



<u>שאלה 1</u>

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 מימוש חדש של quicksort, אשר מקבל פרמטר בוליאני rand. אם rand=True, הפונקציה תבחר את הציר באופן אקראי (כמו בגרסה שראיתם בהרצאה), ואם rand=False הבחירה תהיה דטרמיניסטית (השינויים מודגשים בצהוב):

```
def quicksort(lst, rand=True):
    """ quick sort of lst """
    if len(lst) <= 1:
        return lst
    else:
        pivot = random.choice(lst) if rand else lst[0]
        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 quicksort(smaller, rand) + equal + quicksort(greater, rand)
```

הערה: פקודת ההשמה של pivot משתמשת באופרטור טרנרי (Ternary Operators) – זוהי דרך מקוצרת ונוחה לבצע השמה בפייתון, כתלות בקיום תנאי כלשהו (במקרה שלנו – אם rand=True אנו מבצעים השמה אחת למשתנה pivot, ואם rand=False אנו מבצעים השמה אחרת).

כזכור, זמן הריצה המצופה של quicksort (עם אקראיות) הוא $O(n\log n)$ כאשר n הוא אורך הרשימה, ואילו זמן הריצה הגרוע ביותר הוא $O(n^2)$.

<u>סעיף א'</u>

נרצה לבחון את ההבדל בזמני הריצה בין quicksort האקראית לבין quicksort הדטרמיניסטית על **<u>קלט אקראי</u>**.

ממשו את הפונקציה quicksort_comparison_random_input(n, t) אשר ***<u>מגרילה</u>*** t רשימות אקראיות באורך n המכילות את המספרים $1,2,\dots,n$ (כך שכל איבר מופיע בדיוק פעם אחת), ומחזירה את זמן הריצה הממוצע בשניות של quicksort האקראית ושל quicksort הדטרמיניסטית (בהתאמה, כ-tuple באורך 2) על t הרשימות.

השתמשו בספרייה random כדי להגריל את הרשימות (למשל בעזרת הפונקציה random.shuffle).

הריצו את הפונקציה עם ערכי n, t שונים. פרטו את הממצאים בקובץ ה-PDF והסיקו – מי מהפונקציות עדיפה עבור קלט אקראי?

<u>סעיף ב'</u>

נרצה לבחון את ההבדל בזמני הריצה בין quicksort האקראית לבין quicksort הדטרמיניסטית על **<u>רשימות ממוינות</u>**.

ממשו את הפונקציה quicksort_comparison_ordered_input(n, t) אשר מחזירה את זמן הריצה הממוצע בשניות של quicksort האקראית ושל quicksort הדטרמיניסטית (בהתאמה, כ-tuple באורך 2) על הרשימה $[1,2,\dots,n]$, על פני t הרצות.

הריצו את הפונקציה עם ערכי n, t שונים. פרטו את הממצאים בקובץ ה-PDF והסיקו – מי מהפונקציות עדיפה עבור קלט ממוין?

שאלה 2

סעיף א'

לפניכם שני מימושים שונים לחישוב מקסימום של רשימה באופן רקורסיבי. עבור כל אחד מהמימושים, ציירו את עץ הרקורסיה המתקבל מהרצת הפונקציה על רשימה L באורך n (מומלץ להתחיל בלצייר לעצמכם את העץ המתאים עבור רשימה באורך 3 או 4). נתחו את סיבוכיות זמן הריצה של כל אחד מהמימושים במקרה הגרוע. בפרט, יש לתת חסם עליון הדוק ככל הניתן על זמן הריצה.

i.

```
def max_v1(L):
    if len(L) == 1:
       return L[0]

    mid = len(L) // 2
    first_half = max_v1(L[:mid])
    second_half = max_v1(L[mid:])

    return max(first_half, second_half)
```

ii.

```
def max_v2(L):
    if len(L) == 1:
       return L[0]

    without_left = max_v2(L[1:])

    return max(without_left, L[0])
```

סעיף ב'

במימושים בסעיף א' אנו משתמשים ב-slicing על מנת לחתוך את הרשימה בעת הקריאות הרקורסיביות (תזכורת: פעולת slicing מייצרת רשימה חדשה בזיכרון, ולוקחת זמן לינארי באורך ה slice שנוצר). נרצה לייעל את הפונקציות על ידי החלפת פעולות ה-slicing בשימוש חכם באינדקסים.
ממשו את הפונקציות max_v1_improved(L) ו-max_v2_improved(L) שבקובץ השלד, אשר מבצעות את אותו החישוב של הפונקציות המקוריות (כלומר מוצאות מקסימום ברשימה בעזרת אותן תתי בעיות רקורסיביות כמו בסעיף א') אבל <u>ללא שימוש ב-slicing</u>. שימו לב שלכל פונקציה, עליכם לממשה כפונקציית מעטפת שמבצעת קריאה ראשונית לפונקציה רקורסיבית מתאימה.
תוכלו להוסיף לפונקציות הרקורסיביות קלטים שמייצגים אינדקסים ברשימה.
נתחו את זמן הריצה של כל אחד מהמימושים במקרה הגרוע והשוו אותם לאלו של המימושים בסעיף א'.

סעיף ג'

לסעיף זה אין קשר לסעיפים א' וב' או לפונקציה max.
ממשו את הפונקציה reverse(L) המקבלת רשימה L ומחזירה רשימה חדשה של איברי L בסדר הפוך.

<u>הנחיות</u>:

- על זמן הריצה של הפונקציה להיות $O(n)$ במקרה הגרוע, עבור רשימה L באורך n.
- על הפונקציה שאתם מממשים להיות רקורסיבית, או להיות פונקציית מעטפת שקוראת לפונקציה רקורסיבית.
- עליכם לממש בעצמכם את הפונקציה, ללא שימוש בפונקציות עזר של פייתון שיכולות לבצע את אותה משימה (כגון reverse).

<u>הסבירו</u> מדוע זמן הריצה עומד בדרישת הסיבוכיות.

דוגמת הרצה:

```
>>> reverse([1, 5, "hello"])
["hello", 5, 1]
```

<u>שאלה 3</u>

בתרגול הוצגה בעיית Count Paths. הקלט הינו רשימה L באורך d של מספרים אי-שליליים המייצגת נקודה בשריג d-מימדי. פתרון לבעיה הוא **מספר הדרכים השונות** להגיע מראשית הצירים ([0, 0, …, 0]) ל-L, כאשר בכל צעד מתקדמים ביחידה אחת באחד הצירים. לפניכם קוד לפתרון הבעיה כפי שהוצג בכיתה.

```python
def cnt_paths(L):
    if all([elem == 0 for elem in L]):
        return 1

    result = 0
    for i in range(len(L)):
        if L[i] != 0:
            L[i] -= 1
            result += cnt_paths(L)
            L[i] += 1
    return result
```

הערה: בשאלה זו ניתן להניח כי גישה למילון עולה O(1) זמן (בהמשך הקורס נראה שגישה למילון מבטיחה זמן O(1) רק באופן ממוצע, ולא במקרה הגרוע ביותר, אבל בשאלה זו נתעלם מכך, וגם נוכל להתרשם מזמני הריצה הממוצעים).

i. בתרגול הראינו חסם תחתון של $O(d^n)$ לזמן הריצה עבור הקלט $L = [n, n, \dots, n]$, שהיה מטריד ובלתי שימושי עבור קלטים גדולים. בסעיף זה נשתמש בממואיזציה לשיפור היעילות. ממשו את הפונקציה cnt_paths_mem בקובץ השלד. ניתן (ורצוי) לממש פונקציה דומה לפונקציה הנתונה cnt_paths עם שינוי קטן לתמיכה בממואיזציה. בדקו את הפונקציה על ידי הרצה על מספר קלטים עד שתשתכנעו בנכונותה. ניתן להשוות לפלט הפונקציה הנתונה cnt_paths.

ii. נתחו בקובץ ה-PDF את סיבוכיות זמן הריצה של גירסת הממואיזציה שכתבתם במונחים של $O(\cdot)$.
הנחיה לחישוב סיבוכיות הזמן: שימו לב איך מספר הנקודות בשריג תלוי בערכי n_i השונים. בנוסף, שימו לב לכמות העבודה בכל צומת. האם היא קבועה? תלויה ב-n_i? ב-d?
רמז: תוכלו לנסות לחשוב בתחילה על הבעיה בשני מימדים, להמשיך לשלושה מימדים, ורק אז להכליל ל-d מימדים.

iii. בצעו הרצות על קלטים שונים תוך מדידת הזמנים של גירסת הממואיזציה לעומת הגירסה ללא הממואיזציה. שתפו בקובץ ה-PDF את טבלת הקלטים, זמני ההרצות, והפלטים של ההרצות השונות. כמו כן התייחסו **לקצב גידול** זמני הריצה של שתי הפונקציות ביחס לגודל הקלט.

iv. פתרו את הבעיה ללא רקורסיה וממשו את הפונקציה cnt_paths_iter בקובץ השלד. בידקו את נכונות הפתרון מול cnt_paths_mem. הרחיבו בקובץ ה-PDF, האם הקוד קל להבנה ולבדיקה? האם הוא מהיר יותר מגירסת הממואיזציה? מדוע?
הנחיה: בגירסה הרקורסיבית, כדי למצוא פתרון עבור נקודת שריג מסוימת, חיברנו את הפתרונות של הנקודות שהופיעו לפניה. כאשר גם אלו לא היו ידועות, נאלצנו לרדת בעץ הרקורסיה עוד ועוד כדי לחפש נקודת עוגן עבורה הפתרון כן ידוע. בפתרון איטרטיבי, האתגר הוא למצוא סדר נכון על נקודות הביניים בשריג, כך שמתחילים מראש מנקודה שכן יודעים לחשב, ממשיכים לנקודה נוספת שיודעים לחשב בעזרתה, וכן הלאה. שימו לב שלצורך פתרון איטרטיבי ניתן למצוא סדר למעבר על הצמתים בשריג כך שבכל צומת מבקרים רק פעם אחת – לאחר שכל הדרכים אליו כבר חושבו.

<u>שאלה 4</u>

גרף מכוון $G=(V,E)$ מוגדר על ידי קבוצה של n **צמתים**, נסמנה $V=\{0,1,\dots,n-1\}$, וקבוצת **קשתות** $E\subseteq V\times V$ של זוגות צמתים (שימו לב שהשתמשנו בגרפים מכוונים כדי לתאר רשתות באלגוריתם PageRank, כאשר הצמתים ייצגו אתרים, והקשתות ייצגו לינקים). בהינתן גרף $G=(V,E)$ עם n צמתים, **מטריצת השכנויות A** של הגרף G היא מטריצה בגודל $n\times n$ אשר מקיימת:

$$A_{ij}=\begin{cases}1\ if\ (i,j)\in E\\0\ if\ (i,j)\notin E\end{cases}$$

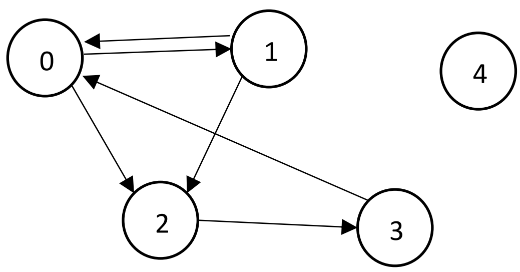
במילים, לכל זוג צמתים $i,j\in V$ מתקיים ש $A_{ij}=1$ אם ורק אם יש קשת בגרף מ-i ל-j.
לדוגמה, עבור הצמתים $V=\{0,1,2,3,4\}$ והקשתות $E=\{(0,1),(1,0),(1,2),(0,2),(2,3),(3,0)\}$, מטריצת השכנויות A היא:

$$A=\begin{pmatrix}0&1&1&0&0\\1&0&1&0&0\\0&0&0&1&0\\1&0&0&0&0\\0&0&0&0&0\end{pmatrix}$$

נייצג את המטריצה בפייתון על ידי רשימה של רשימות, כך שכל תת רשימה תייצג שורה במטריצה. לדוגמה, את המטריצה הנ"ל נייצג בפייתון על ידי הרשימה:

$$A=[[0,1,1,0,0],[1,0,1,0,0],[0,0,0,1,0],[1,0,0,0,0],[0,0,0,0,0]]$$

דרך נוחה לייצג גרף מכוון באופן ויזואלי היא באמצעות ציור הצמתים כמעגלים, וציור הקשתות כחיצים בין המעגלים. למשל את הדוגמה הנ"ל ניתן לצייר באופן הבא:



נאמר שקיים בגרף **מסלול באורך k** בין צומת s לצומת t, אם קיימת סדרה של קשתות $e_1,\dots,e_k\in E$ מ-s ל-t.
באופן יותר פורמלי, אם נסמן $e_i=(v_i,u_i)$, אז לכל $1\le i<k$ צריך להתקיים ש- $u_i=v_{i+1}$, וכן $v_1=s$ ו-$u_k=t$.
באופן שקול, ניתן לחשוב על מסלול כעל סדרת צמתים $v_1,\dots,v_{k+1}$ כך שבין כל זוג עוקב של צמתים יש קשת בגרף. נגדיר מסלול באורך $k=0$ באופן טבעי להיות מסלול ריק (כלומר, יש מסלול באורך 0 בין s ל-t אם ורק אם (אמ"ם) $s=t$), ומסלול באורך $k=1$ להיות קשת בגרף (כלומר יש מסלול באורך 1 מ-s ל-t אמ"ם הקשת (s,t) בגרף).

<u>הערות</u>:

- ההגדרה לעיל היא של **גרף מכוון**. מקרה פרטי של גרף מכוון הוא **גרף לא מכוון** שבו לכל קשת $(u,v)\in E$, גם $(v,u)\in E$. בשאלה זו נעסוק במקרה הכללי יותר, כלומר בגרפים מכוונים.
- לאורך כל השאלה נניח כי בגרפים **אין קשתות עצמיות**, כלומר לכל i מתקיים ש-$(i,i)\notin E$.
- לאורך השאלה נסמן ב-n את מספר הצמתים בגרף, וב-k אורך של מסלול בגרף.

<u>סעיף א'</u>

ממשו את הפונקציה $legal\_path(A, vertices)$ המקבלת מטריצת שכנויות A של גרף כלשהו, ורשימה של צמתים $vertices$ של צמתים בגרף, ומחזירה True אם רשימת הצמתים מהווה מסלול בגרף, ו- False אחרת. ניתן להניח כי הגרף אינו ריק, כלומר קיים בו צומת אחד לפחות.

דוגמאות הרצה (A היא מטריצת השכנויות של הגרף המתואר מעלה):

```
>>> A = [[0,1,1,0,0], [1,0,1,0,0], [0,0,0,1,0], [1,0,0,0,0],[0,0,0,0,0]]
>>> legal_path(A, [0, 1, 2, 3])
True
>>> legal_path(A, [0, 1, 2, 3, 0, 1])
True
>>> legal_path(A, [0, 1, 2, 3, 4])
False
```

סעיף ב'

בסעיף זה נניח ש- $k < n$. נדון בניסיון לממש פונקציה אשר בודקת האם קיים מסלול באורך k בין שני צמתים s, t בגרף.
הפונקציה הרקורסיבית הבאה מקבלת מטריצת שכנויות A, שני צמתים s ו-t, ומספר k, ומחזירה True אמ"ם קיים מסלול באורך k מ-s ל-t:

```
def path_v1(A, s, t, k):
    if k == 0:
        return s == t

    for i in range(len(A)):
        if A[s][i] == 1:
            if path_v1(A, i, t, k-1):
                return True
    return False
```

i. ציירו את עץ הרקורסיה המתאים עבור ההרצה הבאה (מכיוון ש-A ו-t אינם משתנים לאורך הריצה של הפונקציה, רשמו בכל צומת בעץ את הערכים המתאימים של s ו-k בלבד):

```
>>> A = [[0,1,1,0,0], [1,0,1,0,0], [0,0,0,1,0], [1,0,0,0,0], [0,0,0,0,0]]
>>> path_v1(A, 0, 4, 3)
```

ii. הראו שיש קלטים עבורם זמן הריצה אקספוננציאלי ב-n, כאשר n מייצג את מספר הצמתים בגרף (שמיוצג ע"י מטריצת השכנויות). כלומר, הראו שקיים קבוע $c > 1$ (שאינו תלוי ב-n), ושקיים $n_0 \in \mathbb{N}$ כך שלכל $n_0 \leq n$ קיים קלט עם n צמתים בגרף, שזמן הריצה שלו הוא לפחות c^n. **הסבירו** את תשובתכם.
<u>הערה</u>: עבור הקלטים שאתם נותנים צריך להתקיים ש- $k < n$.
<u>רמז (עבה)</u>: לכל n (החל ממקום מסוים) ניתן למצוא דוגמה יחסית פשוטה של קלטים עבורם זמן הריצה של הפונקציה הוא לפחות $(n-2)^{n-2}$.

סעיף ג'

בדומה לסעיף הקודם, גם הפונקציה הרקורסיבית הבאה מקבלת מטריצת שכנויות A, שני צמתים s ו-t, ומספר k, ואמורה להחזיר True אמ"ם קיים מסלול באורך k מ-s ל-t:

```
def path_v2(A, s, t, k):
    if k == 0:
        return s == t

    # ADD YOUR CODE HERE #

    for i in range(len(A)):
        mid = k // 2
        if path_v2(A, s, i, mid) and path_v2(A, i, t, k – mid):
            return True
    return False
```

i. שימו לב שהמימוש לא משתמש כלל במטריצת השכנויות של הגרף כדי לבדוק קיומן של קשתות, ולכן לא יתכן שהוא נכון. תנו דוגמה לקלט שעבורו הפונקציה הנ"ל לא מחזירה את הפלט הנדרש.

ii. תקנו את הפונקציה בקובץ השלד, על ידי **הוספת קוד בחלק המסומן**, וללא מחיקת הקוד הקיים. (שימו לב שלסעיף זה אין בדיקה ב-test שבקובץ השלד כדי לא לחשוף את התשובה *לסעיף הקודם*. הקפידו לוודא את נכונות הפתרון שלכם!)

iii. נסמן ב-$f(n)$ את זמן הריצה של הפונקציה במקרה הגרוע על קלט בגודל n. נאמר ש-$f(n)$ היא **סופר-פולינומיאלית** ב-n אם לכל קבוע c, קיים n_0 כך שלכל $n > n_0$ מתקיים $f(n) > n^c$. הגדרה שקולה היא שלא קיים קבוע c כך ש-$f(n) = O(n^c)$. לדוגמה, הפונקציה 2^n היא סופר-פולינומיאלית ב-n, כמו גם $n^{\log(n)}$.

הראו שזמן הריצה של הפונקציה במקרה הגרוע הוא סופר-פולינומיאלי במספר הצמתים n. כלומר, הראו שקיימת פונקציה סופר פולינומיאלית f, ושקיים $n_0 \in \mathbb{N}$ כך שלכל $n_0 \leq n$ קיים קלט עם n צמתים בגרף, שזמן הריצה שלו הוא לפחות $f(n)$.
<u>**הסבירו**</u> את תשובתכם.
<u>הערה</u>: עבור הקלטים שאתם נותנים צריך להתקיים ש- $k < n$.
<u>רמז</u>: ניתן למצוא דוגמה יחסית פשוטה של קלטים עבורם זמן הריצה של הפונקציה הוא לפחות $(n-1)^{\log(n-1)}$. שימו לב שזו אכן פונקציה סופר-פולינומיאלית.

<u>סעיף ד'</u>

נסה כעת לפתור בעיה מעט שונה. בהינתן A מטריצת שכנויות, וצמתים s, t, האם קיים מסלול **באורך כלשהו** בין s ל-t? לפניכם מימוש <u>לא נכון</u> לפתרון בעיה זו:

```
def path_v3(A, s, t):
    if s == t:
        return True

    for i in range(len(A)):
        if A[s][i] == 1:
            if path_v3(A, i, t):
                return True
    return False
```

i. לפניכם 4 טענות על הפונקציה path_v3:
   a. קיים קלט (A, s, t) כך <u>שיש</u> מסלול בין s ל-t, אך הפונקציה תחזיר False.
   b. קיים קלט (A, s, t) כך <u>שיש</u> מסלול בין s ל-t, אך הפונקציה לא תסיים לרוץ, או שתיזרק שגיאת זמן ריצה.
   c. קיים קלט (A, s, t) כך <u>שאין</u> מסלול בין s ל-t, אך הפונקציה תחזיר True.
   d. קיים קלט (A, s, t) כך <u>שאין</u> מסלול בין s ל-t, אך הפונקציה לא תסיים לרוץ, או שתיזרק שגיאת זמן ריצה.

לכל טענה, קבעו האם היא נכונה או לא. אם היא נכונה, תנו דוגמה לקלט שמוכיח זאת, ואחרת הסבירו בקצרה מדוע לא קיים קלט כזה.

ii. לפניכם מימוש נוסף של הפונקציה, אשר מוסיף משתנה עזר לחתימה. שימו לב שמלבד השימוש בפונקציית מעטפת ומשתנה העזר הנוסף, מימוש זה זהה למימוש של path_v3 (ובפרט, כל קלט בעייתי מהסעיף הקודם יהיה בעייתי גם במימוש החדש). תקנו את הקוד על ידי **הוספת קוד בלבד** (ניתן להוסיף את הקוד בכל מקום, אבל אין למחוק קטעי קוד קיימים). הקפידו שסיבוכיות זמן הריצה של הקוד במקרה הגרוע תהיה $O(n^2)$, כאשר n הוא מספר צמתים בגרף.
<u>**הסבירו**</u> מדוע המימוש שלכם עומד בדרישת הסיבוכיות ומדוע הוא תקין.

```
def path_v4(A, s, t):
    L = [False for i in range(len(A))]
    return path_rec(A, s, t, L)

def path_rec(A, s, t, L):
    if s == t:
        return True

    for i in range(len(A)):
        if A[s][i] == 1:
            if path_rec(A, i, t, L):
                return True
    return False
```

שאלה 5

הנחיות לכל הסעיפים בשאלה זו :

- על הפונקציה שאתם מממשים להיות רקורסיבית, או להיות פונקציית מעטפת שקוראת לפונקציה רקורסיבית.
- ניתן להניח שפעולות אריתמטיות לוקחות זמן קבוע.
- אין צורך להפעיל ממואיזציה

בהינתן רשימה lst לא ריקה של מספרים שלמים וחיוביים השונים זה מזה, ומספר שלם s, נאמר ש**ניתן ליצור את s מ-lst** אם ניתן להגיע ל-s על ידי חיבור וחיסור של איברי lst.

סעיף א'

בסעיף זה, נבדוק האם ניתן ליצור את s מ-lst תחת ההגבלה שאנו משתמשים בכל איבר ב-lst **בדיוק פעם אחת**.
לדוגמה, אם $lst = [5,2,3]$ ו- $s = 6$ אז ניתן ליצור את s מ-lst תחת ההגבלה, מכיוון ש-

$$5 - 2 + 3 = 6$$

כמו כן ניתן ליצור מ-L את $s = -10$ מכיוון ש-

$$-5 - 2 - 3 = -10$$

לעומת זאת, לא ניתן ליצור מ-lst את $s = 9$ או את $s = 7$ תחת ההגבלה.
ממשו את הפונקציה can_create_once(s, lst) אשר מחזירה True אם אפשר ליצור את s מ- lst, ו-False אחרת, תחת הגבלה זו.

מה גודל עץ הרקורסיה כתלות באורך הרשימה?

סעיף ב'

בסעיף זה נממש פונקציה דומה לזו מסעיף א', אבל הפעם נרשה להשתמש בכל איבר ב-lst **לכל היותר פעמיים**. לדוגמה, אם $lst = [5,2,3]$ אז הפעם ניתן ליצור את $s = 9$ תחת הגבלה זו, מכיוון ש-

$$5 + 2 + 2 = 9$$

כמו כן ניתן ליצור את $s = 9$ תחת ההגבלה גם בדרך הבאה

$$5 - 2 + 3 + 3 = 9$$

ממשו את הפונקציה can_create_twice(s, lst) אשר מחזירה True אם אפשר ליצור את s מ- lst, ו-False אחרת, תחת הגבלה זו.

מה גודל עץ הרקורסיה כתלות באורך הרשימה?

סעיף ג'

מומלץ לקרוא על הפונקציה המובנית eval שיכולה לסייע לכם בסעיף זה.
בהינתן רשימה lst של מספרים שלמים חיוביים כמחרוזות וביניהם המחרוזות '+', '*', ו-'-', נחשוב על הרשימה כעל ביטוי מתמטי של חיבור, חיסור וכפל בין מספרים. לדוגמה, הרשימה $['3' , '+' , '2' , '*' , '4', '-' , '6'] = lst$ תייצג את הביטוי :

$$6 - 4 \times 2 + 3$$

בהינתן רשימה lst כזו, ומספר שלם s, נרצה למצוא האם יש דרך למקם סוגריים על הביטוי המתמטי שמייצג lst כך שערך הביטוי בהתאם לחוקי הקדימות של הסוגריים יהיה שווה ל-s. לדוגמה, עבור ה-lst הנ"ל ו-$s = 10$:

$$(6 - 4) \times (2 + 3) = 10$$

כמו כן ניתן להגיע ל $s = 1$ על ידי :

$$\big(6 - (4 \times 2)\big) + 3 = 1$$

ממשו את הפונקציה valid_brackets_placement(s, lst) המקבלת מספר שלם s ורשימה לא ריקה lst כמתואר[1], ומחזירה True אם קיימת השמת סוגריים מתאימה, ואחרת מחזירה False. לשם פשטות הניתוח, נסמן ב-n את כמות המספרים ברשימה lst (כלומר, מספר איברי הרשימה לא כולל הסימנים).

נתחו את גודל עץ הרקורסיה המתקבל מהפתרון שלכם. כלומר, אם הוא פולינומיאלי – הראו חסם עליון. אם גדול יותר – הראו חסם תחתון.

שימו לב : יש פתרון פשוט יחסית, שעבורו גם חישוב גודל עץ הרקורסיה פשוט יחסית, והוא $O(n!)$. ברם, גם פתרונות עם גדלים אחרים של עץ הרקורסיה יתקבלו.

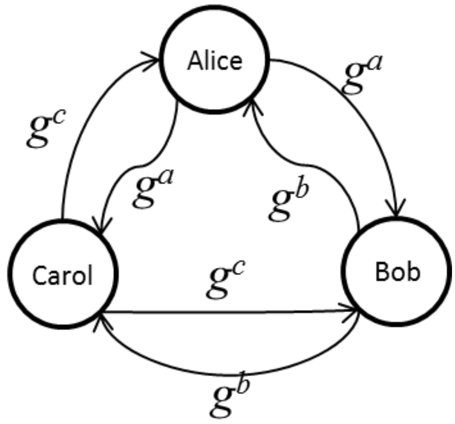
[1] בפירוט : באינדקסים הזוגיים ברשימה (מתחילים מ-0) תמיד יהיו מספרים שלמים חיוביים כמחרוזות, באינדקסים האי זוגיים יהיו אחת המחרוזות '+' , '*', או '-', והרשימה תהיה באורך אי זוגי גדול או שווה ל-1.

<u>שאלה 6</u>

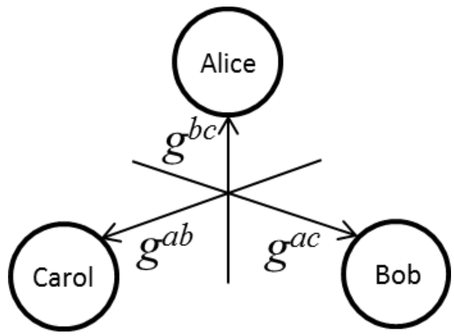
שאלה זו דנה בהרחבת פרוטוקול דיפי-הלמן להחלפת מפתח סודי, ליותר משני משתתפים. נסמן את מספר המשתתפים המעוניינים לייצר להם סוד משותף ב- N.

נזכיר, כי כל החישובים נעשים מודולו *p* ראשוני כלשהו שנבחר מראש. לשם פשטות, בשאלה זו לא נציין זאת בביטויים שבהמשך. כלומר, בכל מקום שכתוב למשל g^a הכוונה היא ל- $g^a \bmod p$.

א. נניח N=3. להלן תיאור פרוטוקול מורחב אפשרי. בשלב הראשון מתבצע הפרוטוקול כפי שלמדנו בכיתה, בין כל זוג משתמשים. לדוגמה, Alice מחשבת את g^a (פעם אחת), ושולחת זאת ל- Bob ול- Carol. תרשים הודעות שנשלחות:



כעת Alice ו- Bob חולקים את הסוד g^{ab}, Alice ו- Carol את הסוד g^{ac}, ואילו Bob ו- Carol את הסוד g^{bc}. כל זוג שולח את הסוד המשותף שלו למשתמש השלישי. למשל, Alice ו- Bob שולחים (אחד מהם או שניהם, לא משנה) ל- Carol את g^{ab}, ובדומה גם שאר הזוגות:



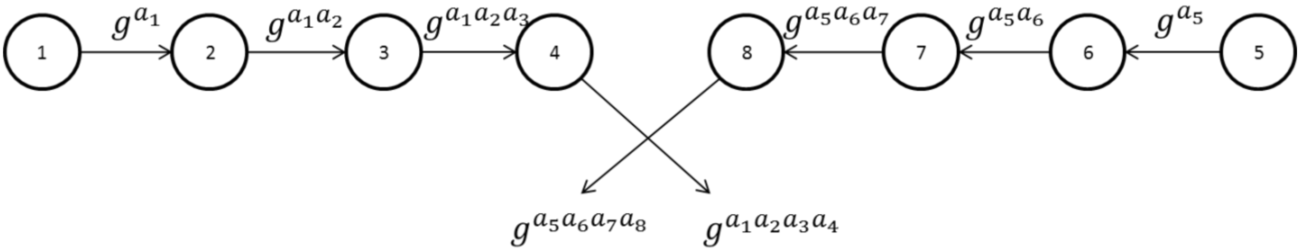
וכל משתמש יכול כעת לחשב את הסוד המשותף g^{abc}.

כמה פעולות modular exponentiation מבצע כל משתמש?

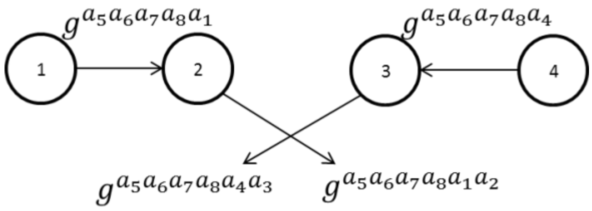
ב. בקובץ ה pdf תארו פרוטוקול דומה, בו כל משתמש מבצע **3** פעולות modular exponentiation בלבד. ציינו אילו הודעות נשלחות בכל שלב. אפשר להיעזר באיור בדומה לאיורים לעיל.

ג. להלן תיאור של פרוטוקול מורחב עבור N=8 משתתפים. נסמן את הסוד הפרטי של משתתף i ב-a_i.

<u>שלב 1</u> : מחלקים את המשתמשים ל- 2 קבוצות שוות, ושולחים את ההודעות הבאות :



שלב 2 : מחלקים כל קבוצה באופן דומה וחוזרים על התהליך, כאשר בכל קבוצה ההודעה ההתחלתית היא ההודעה שנשלחה בסוף השלב הקודם מהקבוצה השנייה. למשל, משתתפים 1, 2, 3 ו- 4 מחולקים שוב וחוזרים על התהליך, עם ההודעה ההתחלתית $g^{a_5 a_6 a_7 a_8}$:



בקובץ ה pdf ענו על ארבע השאלות הבאות :

1) איזו הודעה ישלח משתתף 1 ל- 2 בשלב השלישי (והאחרון)?

2) איזו פעולה יעשה משתתף 2 לאחר השלב השלישי, כדי לחשב את הסוד המשותף?

3) מהו הסוד המשותף לכל 8 המשתתפים?

4) עבור N משתתפים, כמה פעולות modular exponentiation מבצע כל משתתף? יש לתת תשובה במונחים של O, הדוקה ככל שניתן.